Virtual Timing Isolation Safety-Net for Multicore Processors

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

der Fakultät für angewandte Informatik
der Universität Augsburg

eingereicht von

M.Sc. Johannes Freitag

2020

Erstgutachter:	Prof. Dr. Theo Ungerer
Zweitgutachter:	Prof. Dr. Christoph Ament
Tag der mündlichen Prüfung:	09.03.2020

Abstract

Multicore processors promise to offer the performance as well as the reduced space, weight and power needed by future aircrafts. However, commercial off-the-shelf multicore processors suffer from timing interferences between cores which complicates applying them in hard real-time systems like avionic applications. In this thesis, a safety-net system is proposed which enables a virtual timing isolation of applications running on one core from all other cores. The technique is based on hardware external to the multicore processor and completely transparent to the applications, i.e. no modification of the observed software is necessary. The basic idea is to apply a single-core execution based worst-case execution time analysis and to accept a predefined slowdown during multicore execution. If the slowdown exceeds the acceptable bounds, interferences will be reduced by controlling the behavior of low-critical cores to keep the main application's progress inside the given bounds.

Measuring the progress of the applications running on the main core is performed by tracking the application's fingerprint. A fingerprint is created by extraction of the performance counters of the critical core in very small timesteps which results in a characteristic curve for every execution of a periodic program. In standalone mode, without any running applications on the other cores, a model of an application is created by clustering and combining the extracted curves. During runtime, the extracted performance counter values are compared to the model to determine the progress of the critical application. In case the progress of an application is unacceptably delayed, the cores creating the interferences are throttled. The interference creating cores are determined by the accesses of the respective cores to the shared resources. A controller that takes the progress of a critical application as well as the time until the final deadline into account throttles the low priority cores. Throttling is either performed by frequency scaling of the interfering cores or by halt and continue with a pulse width modulation scheme.

The complete safety-net system was evaluated on a TACLeBench benchmark running on an NXP P4080 multicore processor observed by a Xilinx FPGA implementing a MicroBlaze soft-core microcontroller. The results show that the progress can be measured by the fingerprinting with a final deviation of less than 1 % for a TACLeBench execution with running opponent cores and indicate the non-intrusiveness of the approach. Several experiments are conducted to demonstrate the effectiveness of the different throttling mechanisms. Evaluations using a real-world avionic application show that the approach can be applied to integrated modular avionic applications.

The safety-net does not ensure robust partitioning in the conventional meaning. The applications on the different cores can influence each other in the timing domain, but the external safety-net ensures that the interference on the high critical application is low enough to keep the timing. This allows for an efficient utilization of the multicore processor. Every critical application is treated individually, and by relying on individual models recorded in standalone mode, the critical as well as the non-critical applications running on the other cores can be exchanged without recreating a fingerprint model. This eases the porting of legacy applications to the multicore processor and allows the exchange of applications without recertification.

Kurzfassung

Der Einsatz von Multicore Prozessoren in Avioniksystemen verspricht sowohl die Performancesteigerung als auch den reduzierten Platz-, Gewichts- und Energieverbrauch, der zur Realisierung von zukünftigen Flugzeugen benötigt wird. Die Verwendung von seriengefertigten (COTS) Multicore Prozessoren in sicherheitskritischen Echtzeitsystemen ist jedoch sehr komplex, da eine gegenseitige zeitliche Beeinflussung der Anwendungen auf den unterschiedlichen Kernen nicht ausgeschlossen werden kann. In dieser Arbeit wird ein Konzept vorgestellt, das eine virtuelle zeitliche Trennung der Anwendungen, die auf einem Prozessorkern ausgeführt werden, von denen der übrigen Kerne ermöglicht. Die Grundidee besteht darin, eine auf einer Single-Core-Ausführung basierende Laufzeitanalyse (WCET) durchzuführen und eine vordefinierte Verlangsamung während der Multicore-Ausführung zu akzeptieren. Wenn die Verlangsamung die zulässige Grenze überschreitet, wird das Verhalten niedrigkritischer Kerne so gesteuert, dass der Fortschritt der Hauptanwendung innerhalb der Deadlines bleibt.

Die Bestimmung des Fortschritts der kritischen Anwendungen erfolgt durch das Verfolgen eines sogenannten Fingerprints. Ein Fingerprint wird durch Auslesen der Performance Counter des kritischen Kerns in sehr kleinen Zeitschritten erzeugt, was zu einer charakteristischen Kurve für jede Ausführung eines periodischen Programms führt. Ein Modell einer Anwendung wird erstellt, indem die extrahierten Kurven gruppiert und kombiniert werden. Während der Laufzeit werden die ausgelesenen Werte mit dem Modell verglichen, um den Fortschritt zu bestimmen. Falls die zeitliche Ausführung einer ktitischen Anwendung zu stark verzögert wird, werden die Kerne gedrosselt, welche die Störungen verursachen.

Das Konzept wurde mit einem TACLeBench-Benchmark evaluiert, der auf einem NXP P4080 Multicore Prozessor ausgeführt, und von einem Xilinx-FPGA beobachtet wurde. Es konnte gezeigt werden, dass der Fortschritt durch den Fingerprint mit einer endgültigen Abweichung von weniger als 1 % für eine TACLeBench-Ausführung mit laufenden konkurrierenden Kernen gemessen werden kann. Die Evaluation mit einer realen Avionik-Anwendung zeigte, dass das Konzept für integrierte modulare Avionik-Anwendungen (IMA) genutzt werden kann.

Der Ansatz gewährleistet keine robuste Partitionierung im herkömmlichen Sinne. Die Anwendungen auf den verschiedenen Kernen können sich zeitlich gegenseitig beeinflussen, aber ein externes Sicherheitsnetz stellt sicher, dass die Verlangsamung der hochkritischen Anwendung niedrig genug ist, um die Deadlines zu halten. Dies ermöglicht eine effiziente Auslastung des Multicore Prozessors. Außerdem wird jede kritische Anwendung einzeln behandelt und verfügt über ein individuelles Modell. Somit können die kritischen und nicht kritischen Anwendungen, die auf den anderen Kernen ausgeführt werden, ausgetauscht werden, ohne ein Modell neu zu erstellen. Dies vereinfacht die Portierung von bestehenden Anwendungen auf Multicore Prozessoren und ermöglicht den Austausch von Anwendungen ohne eine erneute Zertifizierung.

Acknowledgements

First and foremost, I want to thank my advisers Prof. Theo Ungerer and Sascha Uhrig for their support and guidance. I would also like to thank Prof. Theo Ungerer and Prof. Christoph Ament for rating the thesis. I thank the staff at the System and Networking department at the University of Augsburg for the valuable feedback and discussions during the PhD seminars. Furthermore, I want to thank Dietmar Geiger and Bernd Koppenhöfer for the collaboration, interesting discussions and support with an avionics application which has been used in the evaluation. My gratitude goes to Sascha Uhrig for the extensive discussions, advice and the many things I have learned from him. Likewise, I want to thank Volker Ziegler as well as the complete Communication Technologies team at Airbus Research & Technology for their support and the amazing environment. I would also like to express my gratitude to my family and friends for their continuous support and a great time. Last but not least, I would like to thank my wife Elisabeth for her patience and encouragement during the years of the thesis.

Contents

1. Introduction **1**
- 1.1. Setting the Scene . . . 2
- 1.2. Contribution . . . 3
- 1.3. Organization of the Thesis . . . 4

2. Background **5**
- 2.1. Processors in Avionics . . . 5
  - 2.1.1. Certification . . . 6
  - 2.1.2. Partitioning and Segregation . . . 8
- 2.2. Challenges with Multicore Processors . . . 11
- 2.3. WCET Analysis . . . 12
- 2.4. Guidance on the Use of COTS Multicore Processors . . . 14
- 2.5. Definition of Safety-Net . . . 16

3. Related Work **19**
- 3.1. Interference Reduction and Bounding on COTS Multicore Processors . . . 19
  - 3.1.1. Interference Reduction . . . 20
  - 3.1.2. Interference Bounding . . . 20
- 3.2. WCET Multicore Analysis . . . 21
- 3.3. Scheduling and Mapping . . . 22
  - 3.3.1. Execution Model . . . 22
  - 3.3.2. Adaptive Scheduling . . . 22
- 3.4. Contention-Free Hardware Design . . . 23
- 3.5. Feedback Controllers . . . 24
- 3.6. Fingerprinting . . . 24
- 3.7. Summary . . . 24

4. Safety-Net **27**
- 4.1. Basic Idea . . . 28
- 4.2. Prerequisites / Environment . . . 30
- 4.3. Fingerprinting . . . 30
  - 4.3.1. Which Effects Generate Which Curves . . . 31
  - 4.3.2. Selection of Suitable Performance Counter Events . . . 34
- 4.4. Extraction of Performance Counter Values . . . 35
  - 4.4.1. Possible Safety-Net Architectures . . . 35
  - 4.4.2. Sample Rate . . . 36
  - 4.4.3. Extraction Interface . . . 37
  - 4.4.4. Task Switches . . . 38
- 4.5. Model Creation . . . 40
  - 4.5.1. Recording All Possible Fingerprints . . . 40

4.5.2. Clustering . . . 40
4.5.3. Model Encoding . . . 44
4.6. Interference Detection Algorithm . . . 46
4.6.1. Tracking the Progress of Applications in Real-Time . . . 46
4.6.2. Matching . . . 48
4.6.3. Interference Core Identification . . . 51
4.7. Throttling Techniques . . . 52
4.8. Interference Controller . . . 54

5. Implementation **59**
5.1. Hardware Overview . . . 59
5.2. Observed Multicore Architecture . . . 60
5.3. Safety-Net Processor . . . 62
5.3.1. MicroBlaze . . . 63
5.3.2. Aurora . . . 64
5.3.3. Nexus Unwrapping . . . 64
5.3.4. Ethernet . . . 65
5.4. Extraction of Performance Counter Values . . . 65
5.4.1. Debug Interface Configuration Link . . . 65
5.4.2. Extraction Process . . . 66
5.4.3. Maximum Achievable Read-Out Speed . . . 67
5.5. Model Creation . . . 67
5.5.1. Clustering . . . 68
5.5.2. Model Creation and Encoding . . . 69
5.6. Interference Detection . . . 70
5.7. Controller . . . 72
5.8. Throttling of Interfering Cores . . . 72
5.8.1. Frequency Scaling . . . 73
5.8.2. PWM Halt - Continue . . . 73

6. Evaluation **75**
6.1. Software Executed on the Multicore Processor . . . 75
6.1.1. TACLeBench . . . 75
6.1.2. Real-World Helicopter Application . . . 76
6.1.3. Read/Write Opponent . . . 77
6.2. Environments . . . 78
6.2.1. Hybrid Environment . . . 78
6.2.2. Full System Integration . . . 79
6.2.3. Real-World Application . . . 80
6.3. Evaluation in the Hybrid Environment . . . 82
6.3.1. Sample Rate . . . 82
6.3.2. Performance Counter Events . . . 83
6.4. Evaluation in the Full System Integration Environment . . . 85
6.4.1. Interference Quantification Accuracy . . . 85
6.4.2. Throttling Effectiveness . . . 87
6.4.3. Controller . . . 91
6.4.4. Non-Intrusiveness . . . 97

6.5. Evaluation on a Real-World Helicopter Application 99
6.5.1. IMA Applicability 100
6.5.2. Slowdown Detection 102
6.6. Discussion of the Results 103

7. Conclusion and Future Work **107**
7.1. Conclusion 108
7.2. Future Work 109

Bibliography **111**

List of Figures **121**

List of Tables **125**

List of Acronyms **127**

A. TACLeBench **131**

1. Introduction

Future urban air mobility concepts involve a new generation of small autonomously piloted vertical takeoff and landing aircrafts. Examples of such ultra-light vehicles are Vahana, Pop-up, CityAirbus [5], and Lilium Jet [57] which can transport up to four people and are fully electrically powered. The avionic systems for these kinds of aircrafts need to implement most of the functionality available in current aircrafts while providing additional, more complex and computationally demanding functionality for autonomous flying, such as machine learning and sensor fusion for radar, lidar, and computer-based vision systems. The electronic systems must be optimized for weight, space and power in order to fit into this new generation of aircrafts.

Furthermore, aside from these small aircrafts, improved performance is needed in avionics in satellites, for example for vision-based navigation or image processing in earth observation, as well as in conventional aircrafts due to a growing code base in every new aircraft with enhancements on pilot support systems [10]. In order to allow for more environmentally friendly aircrafts, the reduction of weight and the size of the avionic bay is desirable.

A possible solution, which enables the necessary integration of multiple avionic applications into less avionic computers while providing a performance boost, is the use of multicore processors comprising eight or even more cores. Due to the high integration of processing entities in one system-on-chip, the needed space, weight, and power is reduced compared to separate single-core processors. Communication interfaces are partly substituted by on-chip communication channels, and housing and power supplies are dramatically reduced. Furthermore, the obsolescence problem of single-core processors, which are likely to become rare and expensive, is solved by the usage of multicore processors. However, even though multicore processors are widely used in consumer as well as industrial products, their application in avionic systems is very difficult because airborne systems show special requirements with respect to system reliability and availability because of their safety-critical nature.

An essential requirement for certification is a clear and reliable isolation of safety-critical applications, and this needs to be demonstrated to the certification authorities. One of the major challenges in this context is the interference between applications since theoretically one application can compromise another one running on a different core, at least in the timing domain.

One of the most important issues is the contention on the memory (sub-)system resulting from different applications on the cores since it has a major impact on the actual execution

time of an application. This is based not only on queued accesses to the memory and interconnection systems but also on contention on shared caches. This poses a challenge to the applicability of a Worst-Case Execution Time (WCET) analysis, which was historically conducted to demonstrate timing safety, as it leads to a high overestimation which drastically reduces the efficiency of the processors [69].

A further problem accompanied with multicore processors is the complexity of commercial off-the-shelf (COTS) available chips. These chips usually consist of complex out-of-order execution cores with on-chip devices and an interconnect. The complexity as well as the low availability of documentation complicates the analysis of these processors. Even though first ideas of the regulations on how to apply multicore systems in avionics are presented in the CAST-32 position paper and its follow-up CAST-32A [12], both authored from the Certification Authorities Software Team (CAST), concrete design details are still open.

"We need to develop design techniques that go beyond predictability by design and allow the building of reliable systems from unreliable parts." is stated in the executive summary of the HiPEAC Vision 2017 [20]. That design philosophy is applied in the approach presented in this thesis. The multicore processor is treated as an unreliable part due to its unpredictable timing behavior. It is extended by an external safety-net, e.g. a simple micro controller or FPGA which monitors and controls the execution of applications. This leads to a reliable system while leveraging the performance gain of the multicore processor without the need for understanding and quantifying all the interference channels. In the developed approach, *Fingerprinting* continuously tracks the progress of an application by comparing the current state of execution to a virtual single-core execution of the same application in the external safety-net. Unacceptable timing deviations caused by inter-core interferences can be mitigated by controlling the behavior of the non-critical cores.

1.1. Setting the Scene

Regarding novel technologies, the avionic domain is a very conservative domain, which is mainly caused by possible safety issues. This thesis focuses on the use of multicores with only a single core executing highly (safety) critical applications while the others run applications with lower criticality. This means the first core is executing applications with hard deadlines which must never be missed while the other cores run weakly hard [7], soft, or non real-time applications. Accordingly, a technique that enables performance and timing guarantees for one core on the cost of the other cores' performance is proposed.

The approach focuses on critical applications that are executed periodically, which is typically the case for avionic applications. An example is an application which, in every loop, reads data as input, processes the data and creates an output while the complete procedure happens in a cycle of 5 ms to 100 ms. Any type of algorithm can be computed and the execu-

tion of different code depending on the input is possible in every loop. A lightweight operating system can schedule multiple applications with fixed time slicing (e.g. as in integrated modular avionics (IMA)). The aforementioned restrictions do not apply for the low non-critical applications running on the other cores. However, no timing guarantees can be provided for these applications, and it must be possible to change the timing behavior arbitrarily without crashing neither the high-critical nor the low-critical application.

In order to reuse legacy software, guaranteeing a required performance shall be non-intrusive. Moreover, modifications to an operating system (if any) shall be restricted to a minimum to not increase system complexity too much. As appropriate standards and best-practices propose extra circuits external to the processor system to increase system reliability and safety (e.g. mentioned in [12]), such an external device for guaranteeing performance and timing is targeted. In the optimal case, this timing isolation shall be done in addition to the original tasks of a watchdog system.

1.2. Contribution

The contribution of this thesis is the concept, implementation and evaluation of a timing isolation safety-net for multicore processors consisting of the following parts:

- **Non-intrusive interference quantification** of applications using performance counters for fingerprint creation and tracking. The progress of the critical application can be tracked without the need for modification of the application. In this course, an analysis of suitable performance counter events, possibilities of non-intrusive extraction, an approach for data clustering and model creation as well as an algorithm for the tracking during runtime is developed. The general idea of fingerprinting was presented by Freitag and Uhrig [27].

- **Throttling of interference cores** by influencing the behavior of the low priority cores for interference reduction of the critical core. The throttling interface to the multicore processor as well as frequency scaling and pulse width modulation based on halt and continue are presented.

- **Progress aware controller** that ensures virtual timing isolation between one main application and any other application running on a multicore system utilizing the interference quantification and the throttling techniques in a specifically designed control loop including a discussion on possible safety-net architectures. In parts presented by Freitag and Uhrig [28].

- Evaluation of the **interference quantification accuracy**, the **non-intrusiveness** on the main application, and a comparison of three different **controller algorithms**

regarding fulfillment of timing requirements and utilization of the processor. In parts presented by Freitag, Uhrig and Ungerer [30].

- Analysis of the **applicability** of the timing safety-net approach **to integrated modular avionics (IMA) applications** on a real-world application is demonstrated. Published by Freitag and Uhrig [29].

1.3. Organization of the Thesis

The need for a timing isolation safety-net is further motivated in Chapter 2, where the certification environment and the challenges imposed by multicore processors are described. The evolution of processors in avionics is presented and the current publications of the certification authorities are discussed. Research aiming for solutions to the challenges of multicore processors in avionics are given in Chapter 3. Mitigation techniques applied to COTS components as well as contention free hardware designs are discussed. A statement is given on how the approach developed in this thesis relates to the approaches given in literature.

The concept of the timing isolation safety-net is presented in Chapter 4 where the basic idea and the fingerprinting approach is described. Furthermore, an interference detection algorithm, throttling techniques and an interference controller are explained. An example implementation of this concept is described in detail in Chapter 5 where the safety-net is implemented on an FPGA observing an NXP P4080 multicore processor. Based on this implementation, the different aspects as well as the complete system are evaluated in Chapter 6. The applicability to IMA systems is presented on the example of a real-world helicopter application. Finally, the thesis concludes with Chapter 7 including an outlook on future work.

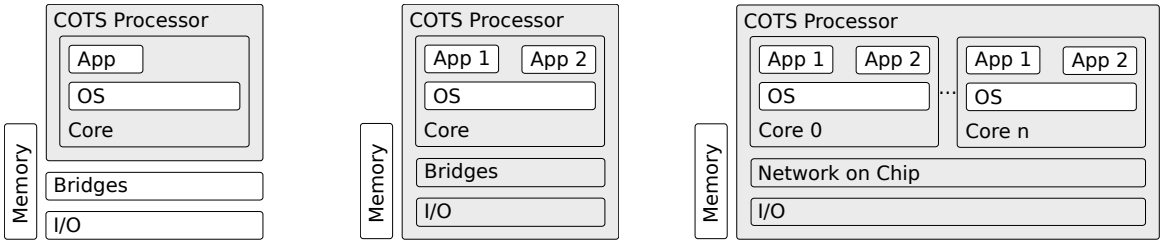
2. Background

In civil aviation, safety is the first priority for the development and qualification of a system. For this reason, several standards were developed to guide the certification process. Some of these standards affect the usage of processing entities and software. However, these standards are not compatible to multicore processors since these processors were not relevant during the creation time of the standards. Therefore, companies and the certification authorities are seeking for ways of applying multicore processors in avionics today. However, these processors imply various challenges compared to single-core processors with respect to safety.

The environment of civil aviation and certification as well as the challenges with multicore processors are discussed in this chapter. The evolution of processors used in avionics and the certification environments including presently applied means for partitioning and segregation are given in Section 2.1. Afterwards, challenges with multicore processors regarding predictability and the effects on WCET analyses are explained in sections 2.2 and 2.3. In Section 2.4 the current publication of the certification authorities on the use of COTS multicore processors is presented. Finally, the definition of the term *Safety-Net* in this thesis is given in Section 2.5

2.1. Processors in Avionics

The evolution of processors used in avionics is shown in Figure 2.1. In the past, a federated architecture was developed for aircrafts, e.g. the Airbus A320 [19]. In this architecture, each



(a) Federated architecture (b) IMA (c) Multicore processor

Figure 2.1.: Evolution of processors in avionics. The past, present and future (from left to right). The gray elements are COTS components while the white hardware components are specifically designed for the individual system. The complexity of the COTS devices is growing from past to future.

computer hardware is uniquely designed for the specific subsystem. For example, the fuel management software is executed on a specific processor that fulfills the requirements of this application. The interfaces of the computer are individually designed according to the needs of the system. This is shown in Figure 2.1a. The actual processor is a COTS component while the other components are specifically designed for each subsystem. However, this approach has many disadvantages. One of the drawbacks is the poor re-usability of existing software, obsolescence mitigation, and the maintainability of these computers. In case of a hardware fault, the specific computer has to be replaced. The most severe weak spot of this architecture is the huge consumption of Size, Weight and Power (SWaP), since every computer needs dedicated wiring, power supplies, etc. With the design of new aircrafts (e.g. the Airbus A380) which implemented more and more functionalities, this was no longer tolerable [41].

Therefore, the concept of Integrated Modular Avionics (IMA) (described in [80]) is implemented in current aircrafts such as the Airbus A380, A350, and Boeing 787. The concept aims to integrate multiple functions on one computer to reduce SWaP. Avionics shall consist of modular computers with standardized interfaces which are not designed for a specific purpose and thus, can be easily replaced. Furthermore, the computers are connected by a network (e.g. AFDX) rather than by discrete lines. Since the processing demand was increased by the need for execution of multiple applications on one processor, the micro controllers used in the federated architecture were replaced by more powerful COTS single-core microprocessors (see Figure 2.1b). There, the complexity of the processor is increased as bridges and simple I/O are already implemented in the processor.

To further increase the integration and to satisfy the processing demands of future applications, the use of COTS multicore processors as shown in Figure 2.1c are the next evolution. This allows for the execution of multiple IMA applications on multiple cores and also for the execution of parallel applications. These COTS processors are very complex as they consist of multiple processing cores, many integrated devices and I/Os, and a complex interconnect. Thus, certification is complicated since in addition to the assuring that no IMA partition is influencing other partitions on the same core, it has to be ensured that no application on other cores is influenced.

2.1.1. Certification

The purpose of certification in civil avionics is to document that a given device meets all applicable regulatory requirements [64]. In the certification process, an applicant (e.g. an aircraft manufacturer) defines a product, determines the applicable regulatory requirements, and demonstrates to the certification authorities that the requirements have been met. The certification authorities are different from country to country. For example, the certification authority in the USA is the Federal Aviation Administration (FAA) while the counterpart in Europe is the European Aviation Safety Agency (EASA).

Software or hardware cannot be individually certified; only systems that might or might not include hard/software can be certified. Hence, multiple standards are to be considered. In the following, the most relevant guidelines for the use of processors and software in avionics are explained. Most of the guidelines were jointly developed by the Radio Technical Commission for Aeronautics (RTCA) and the European Organization for Civil Aviation Equipment (EUROCAE) and are identically considered by the FAA and EASA.

- ARP4754A - Guidelines For Development Of Civil Aircraft and Systems [84] is a guideline dealing with the system development plan that defines the measures depending on the assurance level of the respective system. It was recognized by EASA and FAA [24].
- ARP4761 - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment [83] is a complementary guideline to ARP4754A which aims for the safety plan.
- DO-178B/C - Software Considerations in Airborne Systems and Equipment Certification [81] describes objectives for software planning, development and verification. The number of objectives to be fulfilled depends on the DAL of the application. The goal of being compliant to this standard is to provide confidence that the respective software is safe.
- DO-254 - Design Assurance Guidance for Airborne Electronic Hardware [79] provides guidance on the development of electronic hardware used in avionics. This does not only cover ASICs but also FPGAs, circuit boards, etc. The standard recommends activities to be performed during the development and is the hardware counterpart to DO-178C.
- DO-297 - Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations [80] describes means for the development, installation and verification of safe IMA systems as described in Section 2.1. This includes hardware as well as software aspects.

The relations between these documents as they as described in ARP4754A [84] are shown in Figure 2.2. ARP4754A defines the overall system development plan which is complemented by the safety plan described in ARP4761. The interface between the ARP4754A and the more software/hardware specific guidelines are the requirements to the individual systems. These are defined and verified in the ARP4754A process. The process described in DO-178C/DO-254 assumes that the given requirements are correct and complete. Verification in DO-178C/DO-254 considers whether the developed hard-/software meets the requirements. The DO-297 standard interfaces both, the DO-178C and DO-254, as IMA measures affect both domains.

In all the mentioned guidelines, the recommended actions depend on the design assurance levels (DAL) of the respective system. These levels are consistently defined in the standards

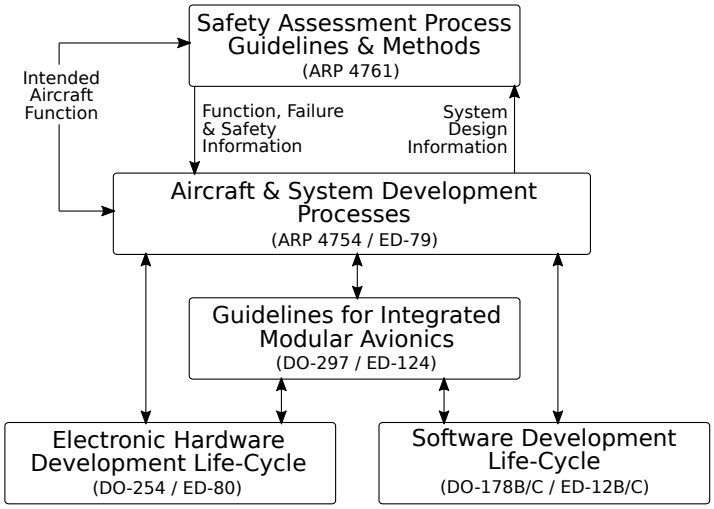


Figure 2.2.: Relations between the different guidelines relevant for the certification of systems which include processors and software [84].

DAL	Severity	Probability	Potential Effects on Passengers or Cabin Crew
A	Catastrophic	$< 10^{-9}$	Multiple fatalities
B	Hazardous	$< 10^{-7}$	Serious or fatal injury to a relatively small number of the occupants other than the flight crew
C	Major	$< 10^{-5}$	Physical distress to passengers or cabin crew, possibly including injuries
D	Minor	$> 10^{-5}$	Some physical discomfort to passengers or cabin crew
E	No Effect	N/A	None

Table 2.1.: Design Assurance Levels (DAL) according to [90]. The probability defines the chance of failure per flight hour.

as shown in Table 2.1. Systems are categorized into these levels depending on the potential effects on the passengers. The highest level is DAL-A. The number of potential failures of such a system has to be lower than 1 per 10^9 flight hours. Thus, the higher the DAL, the higher the effort for development and certification of the respective system. One example of different recommended actions during software verification is the code coverage analysis guidelines in DO-178C. While for DAL-A modified condition/decision coverage (MCDC) is recommended, for DAL-B only decision coverage is required [25].

2.1.2. Partitioning and Segregation

In order to integrate multiple applications of different software levels on the same IMA hardware, an operating system has to provide an abstraction layer as well as robust partitioning to ensure that applications are not disturbed by other applications. Even though the concept of

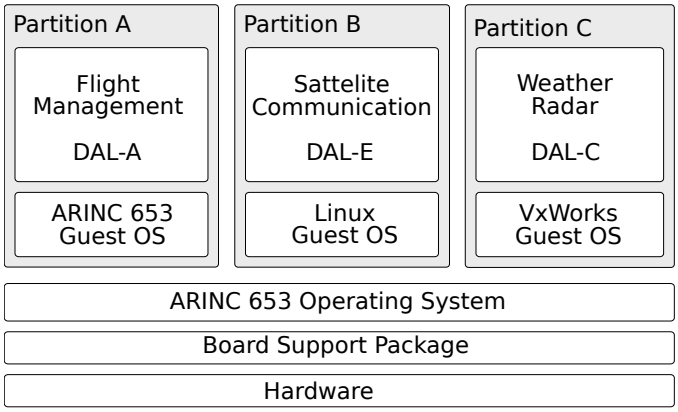


Figure 2.3.: Example of an Integrated Modular Avionics (IMA) system on a single-core processor. Three partitions which include applications of different DALs co-exist on one processing core separately from each other. Partitioning is ensured by the ARINC 653 operating system.

partitioning is universal, the dominant software specification for partitioning in the avionics industry is the ARINC 653 (Avionics Application Standard Software Interface) which was defined by Aeronautical Radio, Incorporated (ARINC). This standard defines partition, process and time management as well as communication channels between partitions and error handling. This way, multiple vendors can provide software to be executed on the same processor. An API called APplication EXecutive (APEX) is described in ARINC 653 for this purpose.

A partition within ARINC 653 is defined as a separate address space. The software within such a partition can be a single thread application, but it can also be an operating system which schedules several threads and runs on top of the underlying ARINC 653 kernel. The choice of operating system within a partition usually depends on the DAL of the individual application. The development of the ARINC 653 kernel is crucial as this component has to ensure the partitioning. In the case of a fault within a partition, the kernel has to ensure that no partition can neither modify another partition's memory nor influence the timing of the other partitions [77].

An example of such a configuration is shown in Figure 2.3. Every partition is independent and has a separate guest operating system running on top of the ARINC 653 operating system which ensures the partitioning. For the highest level of criticality, an ARINC 653 guest operating system is commonly used. For lower DALs, more complex operating systems, such as VxWorks [99] or Linux, can be used in the partitions.

Partitioning can be further divided into spatial and temporal partitioning. Spatial partitioning is the separation of the individual partitions' memory from each other. This is implemented via memory protection units (MPU) or memory management units (MMU) depending on the hardware implemented in the processor. For spatial segregation, not only

Partition	Execution Time	Period
A	10	25
B	5	25
C	10	50
D	10	100

Table 2.2.: Example of an IMA partition schedule.

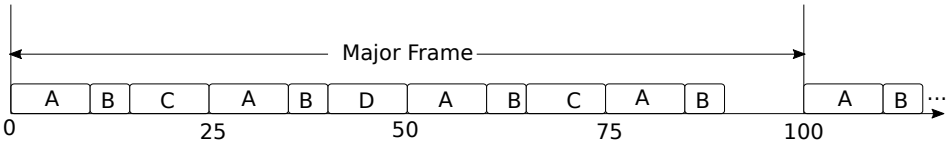


Figure 2.4.: Illustration of the partition schedule given in Table 2.2. The major frame is 100 units long.

the processor cores have to be considered but also I/O devices with direct access to memory. It has to be ensured that no DMA controller can modify the memory region of a partition unintentionally. Communication between partitions is only possible by functions provided by the APEX.

Temporal partitioning ensures that the timing behavior of one partition does not influence the timing of the other ones. Furthermore, every partition is executed for a guaranteed amount of time. It also ensures that if one application misses its deadline, the executed partition is interrupted and the other partitions are not influenced by this misbehavior. In ARINC 653, time slots are assigned to the individual partitions. These time slots are enforced in a cyclic static schedule. A major frame is composed out of the time frames per partition depending on the needed period and the execution time.

An example IMA schedule is given in Table 2.2 with the four partitions A to D and the respective execution time and period. A possible major frame composition for these partitions is shown in Figure 2.4. The major frame is 100 time units long and is not altered during the runtime of the system. Partitions A and B are scheduled every 25 units for their respective duration. Partition C is scheduled two times during the major frame while D is only scheduled once. If there are dependencies between the partitions, e.g. partition D should be scheduled before C, these could be arranged accordingly. From time 90 to 100 the processor is idle to enforce the period of partition A. The major frame is repeated until the system is shut down.

On a single-core processor, time partitioning provides an effective isolation of the timing of the individual partitions. However, the timing can still be influenced via the caches. Depending on the amount of memory accessed by the other partitions, the caches can still have relevant data when a partition is rescheduled, or the data is completely flushed, which leads to a longer execution time. However, the penalty is still small as this can only happen

at transitions from one partition to another and can easily be upper bounded.

On multicore processors, similar partitioning concepts can be applied. However, there are more options on how to schedule partitions on the different cores in parallel. For example, symmetric multiprocessing (SMP) could be applied where different threads of one partition are scheduled in parallel on the different cores. In this case, one major frame is applied to all the cores simultaneously. This approach could be used for newly developed parallel applications which need the full performance of the processor. Another configuration is the asymmetric multiprocessing (AMP) approach. In this case, every core is treated individually. A specific IMA schedule is applied to each core with a different set of applications running on the individual cores. This approach might be usable for porting legacy applications to individual cores of a multicore in order to increase the integration. A mixture of both configurations is also possible. A more detailed discussion on these approaches can be found in [66]. In this thesis only AMP is considered, because the main use case for the presented safety-net approach is the consolidation of legacy applications on multicore processors. In contrast to single-core processors, effective timing isolation by partitioning is a difficult task and poses challenges for certification, as explained in the following sections.

2.2. Challenges with Multicore Processors

For certification, one of the goals is to ensure robust partitioning of the different applications. Spatial partitioning can be achieved similarly to single-core processors with the use of MMUs on the individual cores. However, timing isolation is very hard to achieve on current COTS processors due to shared resources. An example multicore architecture with four cores is shown in Figure 2.5. A level 1 cache is assigned individually to each core, while the level 2 caches are shared between two cores. Furthermore, an interconnect is shared between all the cores and provides access to the memory and I/O. In case all the cores want to access the memory at the same time, three of the cores have to wait since the memory can handle only one request at a time. One core has to wait until the requests of all other three cores are handled. In the worst case, this could be even longer since the arbitration mechanism of the interconnect is usually not disclosed by the chip vendors as it is proprietary [37]. This causes the waiting core to stall even longer, which leads to a delayed execution of the application. Furthermore, there is also congestion on the shared caches, I/O, SoC level configuration registers and other shared blocks on the device, which adds additional sources of conflict. A list of mechanisms effecting the temporal determinism is given in [53]. This competition for shared resources and its alteration of the processor's behavior seen by software on one core is referred to as interference [8]. Due to the complexity of the available COTS devices, not all the possible sources of interference are obvious and quantifiable [67].

The multicore architecture implemented in current COTS devices is optimized for fast

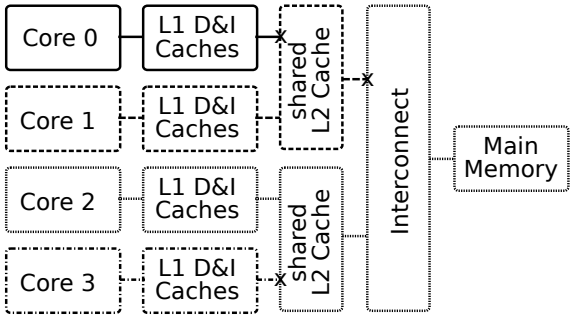


Figure 2.5.: Example for interference channels on a multicore architecture [68]. In case one core accesses the memory, the requests from the other cores that want to access the memory at the same time are waiting.

calculations and fast data transfers needed in servers or mobile devices [12]. The average throughput achieved is high but so is the jitter of the individual accesses. This is not a problem for mobile devices where it does not matter if an application finishes 1 ms later. For embedded applications like avionics, predictability is very important. Since the market for avionics computers is very small compared to consumer devices, chip vendors are not interested in creating a predictable multicore design for comparable prices.

The timing unpredictability due to multiple on-chip shared resources is not the only challenge for certification compared to single-core processors. Another issue is the inability to observe the internal operation, e.g. tracing parallel programs and inject faults into the communication between cores [37]. Furthermore, to allow for a high flexibility, the number of configuration registers per microprocessor has grown significantly [61]. The registers that could have an influence on the performance of the cores, such as frequency scaling, cache configuration, and enabling/disabling of devices like DMA, have to be identified and the state has to be maintained during the whole runtime, even in the case of a single event upset caused by radiation. Furthermore, it has to be ensured that the software accessibility to device configuration during system operation is performed only by the operating system.

2.3. WCET Analysis

In a safety-critical system, the reaction on an input has to follow in a defined time given by the constraints of the system. One example is the pilot's input to gain altitude. This command has to be converted to an input value for the elevator in a predefined time-frame and transmitted to the actuator. In order to show that the algorithm which is computing the new angle finishes in the time-frame on the specific hardware platform, a Worst-Case Execution Time (WCET) analysis can be performed. The WCET analysis results in an estimate of the execution time of an algorithm in the worst-case. The worst-case depends on the input value and the resulting program path but also on the mechanisms in the processor.

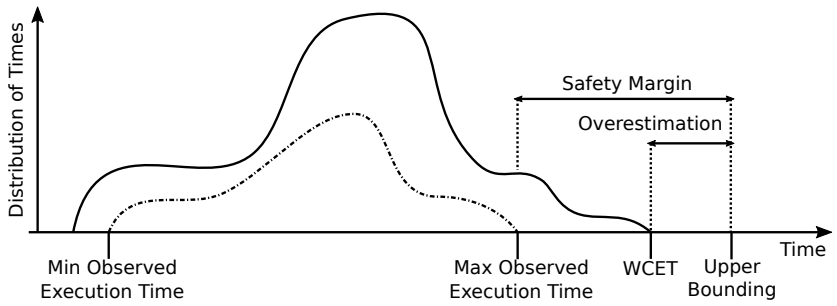


Figure 2.6.: Possible versus observed distribution of execution times of an application [10]. The dashed line represents the distribution of the observed times while the solid line shows the distribution of possible execution times.

This analysis is used to convince the certification authorities that a system implementing this algorithm fulfills the reaction time constraints.

The WCET analysis extends the halting problem which determines whether a task will terminate at some point in time or run forever with the property of *when* a program will terminate. Similar to the halting problem, it is in general not possible to obtain upper bounds on execution times [92, 98]. However, when applying only a restricted form of programming which guarantees that programs always terminate, where for example recursion is not allowed or explicitly bounded and the iteration counts of loops are upper bounded, it is possible to determine a WCET estimate [98].

The WCET analysis can be categorized into two methods: static and measurement based. In a static WCET analysis the code is analyzed without executing it on hardware or a simulator. A control flow graph is generated to determine the longest path within the program. This graph is combined with a precise model of the executing processor architecture. In order to determine a precise WCET estimate, the architecture details such as core pipeline, branch prediction, caches, and memory hierarchy have to be known and modeled in detail.

In a measurement based WCET analysis, the program is executed on the target hardware or simulator with a set of input parameters. This set is potentially incomplete and may not include the input values of the longest path [50]. Furthermore, a complex processor might behave differently for the same input parameters depending on the other devices on the processor and the worst-case input parameter might not lead to the actual WCET.

Figure 2.6 shows a distribution of observed (dashed line) and possible (solid line) execution times. The WCET is unknown since the maximum observed execution time does not necessarily represent the WCET. In order to provide a worst-case guarantee, the execution times are upper bounded by a safety margin. This upper bounding must never be smaller than the actual WCET in order to prevent deadline misses. However, since the actual WCET is unknown, it can only be guessed. This involves the risk of either assuming a too low WCET which leads to a potentially unsafe system or overestimating the WCET which leads to a low

utilization of the processor.

For simple single-core processors, it is possible to perform a static WCET analysis since a simple processor architecture with an in-order execution pipeline, simple memory hierarchy can be modeled precisely. However, on current COTS multicore processors, static analysis cannot be performed because the complexity is too high for the creation of a precise model and documentation is not always available [42]. For a measurement based analysis, there are two main approaches. The first approach is to analyze all the applications on the different cores in parallel which would lead to the most precise results but is infeasible due to the high complexity [96]. The other approach is to treat each core individually and account for the interferences. This is technically feasible, but the maximum observed execution time is very high when assuming the other cores execute opponent threads that only access memory. Furthermore, this execution time is still lower than the actual WCET since the documentation is not given and the hardware most probably cannot be triggered to behave in the worst-case manner. This leads to a very low utilization of the processor as [11] states that "performance loss for worst-case that can go far beyond the expected performance gain of using a multicore processor", which is also confirmed by [69].

2.4. Guidance on the Use of COTS Multicore Processors

The guidelines for certification presented in Section 2.1.1 have to be applied to multicore processors but no specific guidance on how to handle COTS multicore processors is given in these documents. This is because multicore processors were not a relevant architecture at the time these documents were created. However, the Certification Authorities Software Team (CAST) has released a position paper on the usage of multicore processors (CAST-32A) [12], which was published in November 2016. CAST is an international team of certification authority representatives from North America, South America, Europe and Asia. This group has the goal of harmonizing the certification positions of the different countries on software and complex hardware in avionics [90]. Even though it is not an official policy or guidance, it is presenting the current certification authorities' view on multicore processors.

CAST-32A defines a list of objectives to be fulfilled when certifying an application that uses a multicore processor (see Table 2.3). It is depended on the design assurance level if the respective objective should be fulfilled or can be ignored. For a DAL-A application, CAST-32A recommends fulfilling all the objectives when approaching a certification authority.

The planning objectives PL_1 and PL_2 are generic and similar to single-core processors but with the extension of shared resources. These objectives shall ensure that the applicant has studied the architecture of the given processor and is aware of all the features on the SoC that could influence the execution. The resource usage objectives RU_1 and RU_2 target the configuration and the maintaining of the configuration. This does not aim for the timing

	ID	Description
Software Planning	PL_1	**Include MCP specific planning details in the SW plan doc.** Specific processor, number of active cores, software architecture, dynamic software features, whether it hosts an IMA-like system (with applications from different systems) or not, Robust Partitioning supported or not, methods and tools for development and verification.
Planning and Setting Resources	RU_1	**Determine configuration settings that enable to satisfy the functional, performance and timing requirements.**
	RU_2	**Critical configuration settings shall be static and protected against unintended modifications.**
	PL_2	**Include a high level description of shared resource usage and active dynamic hardware features in the hardware and software planning documents.** Intended shared resource allocation and verification to prevent resource capabilities from being exceeded.
Interference Channels and Resource Usage	RU_3	**Identify interference channels and verify the chosen means of mitigation.** Interferences caused by shared memory, shared cache, interconnect, shared I/O or any other shared resource.
	RU_4	**Identify available resources in the intended final configuration, allocate them to the applications and verify that the demands do not exceed the available resources (under worst-case scenarios).**
Software Verification	SWV_1	**Verify that all software components function correctly and have sufficient time when all the software is executing in the intended final configuration.** Depends on the platform classification: 1. Platforms with Robust Partitioning: SW verification and WCET analysis can be done separately for each SW app. 2. All Other Platforms: If interference is mitigated for any software component or set of requirements, the verification of such components can be done separately. Otherwise, verification and WCET analysis shall be done with all software components executing together.
Error Handling	EH_1	**Identify effects of failures that may occur within the MCP plan, design, implement and verify means (which may include a 'safety net' external to the MCP) to detect and handle those failures.**

Table 2.3.: Summary of CAST-32A objectives [3, 12].

isolation but for the complexity of these devices. For example also in the case of an SEU, the correct configuration shall be maintained. RU_3 and RU_4 address the interference channels. These are software or hardware channels through which the applications executed on the cores could interfere with each other. The position paper recommends to identify all the interference channels and find proper mitigation. It states that the demands for resources that are allocated to a specific task shall not be exceeded even under worst-case scenarios. As mentioned in Section 2.2, this is a very hard task and can only be fulfilled if the complete documentation is available, which is usually not the case for COTS components because some details are proprietary.

Objective SWV_1 addresses software verification. It recommends that the software components have sufficient time for execution in the intended final configuration. This is also the case for single-core processors where this is tackled by a WCET analysis as described in Section 2.3. The CAST position paper further distinguishes between multicore processors with and without robust partitioning. Robust time partitioning (on a multicore processor) is defined in CAST32-A as *"achieved when, as a result of mitigating the time interference between partitions hosted on different cores, no software partition consumes more than its allocation of execution time on the core(s) on which it executes, irrespective of whether partitions are executing on none of the other active cores or on all of the other active cores."* For robust partitioned platforms a WCET analysis can be done separately for every application while for other platforms, CAST recommends to perform the WCET analysis on all the software components executing in parallel. As mentioned in the previous section, this is a very hard task and leads to low utilization of the complete processor.

A thorough review of CAST-32A is given in [3] including an application study on the NXP P4080 multicore processor. In this paper, interference channels are identified, and mitigation measures are defined as proposed in CAST-32A. However, the authors conclude that it is not possible to identify all the inference channels due to the lack of documentation. Furthermore, for some interference channels, such as the interconnect, it is not possible to find a suitable mitigation to ensure robust partitioning by using only the configuration possibilities of the processor without an external safety-net.

2.5. Definition of Safety-Net

In the literature, there are different definitions and understandings of the term "Safety-Net" in the context of avionics. A very generic definition is given in CAST-32A [12]: *"A safety net is defined as the employment of mitigations and protections at the appropriate level of aircraft and system design to help ensure continuous safe flight and landing. The safety net methodology focuses on the assumption that a microprocessor will misbehave. The safety net approach is a means to mitigate the risks associated with COTS microprocessors via both*

passive and active methods designed into aircraft systems. This approach requires the safety net to be designed as a function within the aircraft system. The safety net can include passive monitoring functions, active fault avoidance functions, and control functions for recovery of system operations.". In this definition, the safety-net can be any software or hardware that mitigates the misbehavior of a microprocessor. In [55], safety-nets are categorized in *Device Level Safety Nets*, such as on-chip capabilities like lockstep or error correcting codes, and *System Level Safety Nets* which operate on the layer above the SoC, such as external watchdogs or dissimilar architectures. Green et.al. [37] define a safety-net as *"employment of mitigations and protections at the appropriate level of aircraft and system design to help ensure continuous safe flight and landing"*. This paper concludes that in the future ensuring safety at the device level alone is impracticality when microprocessors are used. This is also supported by [61].

In this thesis, external safety-nets are further categorized into **fault detection/mitigation** and **timing error detection/mitigation safety-nets**. Fault detection safety-nets are for example instances that validate if the output values of the processor are in a predefined range or if the voltage levels are correct. Other typical applications are regular checks if the configuration registers still hold the planned value or if it was altered by software or radiation (an overview is given in [35]). In the context of multicore processors, a timing error detection/mitigation safety-net is defined as entity that ensures timing isolation of the individual cores. It is able to detect timing violations caused by interferences on the SoC and can actively execute countermeasures. Some faults may manifest in timing issues, but it cannot be assumed that every fault has an effect in the timing domain and thus, can be detected by a timing safety-net. The safety-net approach developed in this thesis is a timing isolation safety-net.

3. Related Work

Multicore systems in avionic applications are still not widespread. One reason is the difficulty to obtain suitable Worst-Case Execution Time (WCET) estimates since application performance can drop significantly if multiple cores (i.e. applications) share bus and memory as demonstrated by Nowotsch and Paulitsch [69] and Bin et al. [11]. Furthermore, it is not possible to identify all interference channels on COTS multicore processors as presented by Agirre et al. [3] and Mutuel et al. [68]. A WCET analysis on possible worst-case scenarios leads to a high WCET overestimation for current COTS MPSoCs. Hence, the performance gain of the multicore is neglected. Models of the slowdown used for prediction of interferences were studied by Subramanian et al. [91] and Bieber et al. [9]. A model for one of the main interference resources, the RAM and the memory controller, was presented by Kim et al. [47].

In this chapter, an overview of approaches that tackle the timing challenges of the safe application of multicore processors in real-time systems is given. Ideas for limiting the interferences are presented in Section 3.1 and approaches for lowering the overestimation of a WCET analysis are given in Section 3.2. Solutions that base on the scheduling and mapping of applications to the cores, contention free hardware design and the application of controllers for keeping the timing of an application are discussed in sections 3.3, 3.4 and 3.5. Previous usages of performance counters for creating fingerprints are presented in Section 3.6. Finally, a summary is given in Section 3.7.

3.1. Interference Reduction and Bounding on COTS Multicore Processors

There exist several approaches to limit or even control the interferences between the cores on COTS multicore systems. The approach of interference reduction and upper bounding of the interferences that cannot be reduced is often combined in order to provide a full solution.

One example is the Single Core Equivalence (SCE) technology package presented by Sha et al. [88]. This package consists of approaches to minimize DRAM access conflicts, DRAM bandwidth management (MemGuard), and cache management. The authors claim that this technology preserves the constant worst-case execution time assumption, which means that the static and dynamic WCET analysis can be performed for every core individually. However, this is achieved at the cost of a low performance of the multicore processor.

3.1.1. Interference Reduction

Analyses show that the main sources of interferences are shared caches, memory and the interconnect. One popular approach to treat shared caches like private caches is *cache coloring*. With that approach, data and instructions used by different cores are stored in different sets of the cache. This allocation is handled by the operating system which repositions the task memory pages. Cache coloring is a common approach, for example presented in [56]. However, an implementation that targets multicore real-time systems involving cache coloring with extensions is presented as *Colored Lockdown* by Mancuso et al. [62, 51] and Ward et al. [95].

The interference caused by memory accesses can be reduced by assigning cores to different memory banks as shown by Yun et al. [105]. The authors present a technology named *PALLOC* which is an operating system feature that allocates different applications to different memory banks.

A stricter approach is presented by Jean et al. [43, 44]. It involves cache locking with the addition of a TDMA scheme that is applied for accessing the shared resources. Every core is assigned to a TDMA slot in which it is able to access the memory. This is realized by the operating system through MMU reprogramming. The individual accesses are almost contention free, but the performance of the multicore is heavily reduced.

Kim et al. [49, 48] elaborate on the need for isolating not only critical tasks from each other but also the operating system from critical tasks. The proposed approach relies on a modified memory, I/O buffer and device management implemented in the OS.

All the approaches reduce interference spikes, which supports the WCET analysis. However, they do not provide full temporal isolation. Furthermore, the increased predictability is achieved at the cost of a lower average performance of the multicore processor. There was no approach developed for tackling the interferences caused by the common interconnect, and the absence of documentation provided by the chip vendor makes this a very hard task. The isolation techniques are implemented by means in the operating system running on the processor.

3.1.2. Interference Bounding

The interference bounding approaches assign budgets, e.g. of memory accesses or accesses to the common interface to a task/partition. Once the budget is exceeded, the task is suspended until the budget is refilled. This happens periodically. Therefore, it is statically known in the system how many memory accesses are performed by the other cores in the worst-case, and the interference can be upper bounded. The result is a reduced pessimism of the WCET analysis and a lower overestimation.

Memguard is an example of a bounding approach presented by Yun et al. [106]. It relies on the performance counters in each core to count the number of accesses to the shared resources.

Every core is equipped with a bandwidth regulator that guarantees a certain memory access bandwidth to the applications on that core. If a core does not need the full bandwidth, the unused budget can be allocated to a concurrent task.

A similar approach was developed by Nowotsch et al. [70, 71]. Quality of service extensions are implemented in the operating system where each task is assigned a memory budget. Idle times are more efficiently used due to a recalculation of the times required to finish the concurrent tasks once a task has exceeded its budget.

In the presented approaches, robust partitioning is not ensured in the conventional meaning. However, the interference can be upper bounded without drastically reducing the utilization of the multicore processor. The approaches require to evaluate the needed bandwidth for each time slot and to determine a guaranteed bandwidth value. These techniques are interesting for newly developed applications, but they are not suitable for combining multiple legacy single-core avionic applications on a multicore processor because the legacy applications or the underlying operating system would have to be modified, which leads to a high effort in certification (because of increased system complexity).

3.2. WCET Multicore Analysis

A WCET analysis approach to overcome the inability of creating a full model for a static WCET analysis is the hybrid WCET analysis presented by Wegener [96]. In this approach, measurements based on the trace recording, presented by Dreyer et al. [17], are combined with a static analysis. A similar approach is used by RapiTime [82], a tool developed by Rapita Systems Ltd., that also estimates the WCET using measurements. A pure measurement based WCET analysis approach is usually not possible due to the huge amount of traces that has to be recorded from all the cores running in parallel. Thus, measurement snippets are recorded and combined in a static analysis afterwards. Even though, this makes a WCET analysis possible, the main drawbacks accompanied with measurements are the same. It is not guaranteed that the worst-case was recorded because either the input set might not be complete or hardware effects that lead to the worst-case were not triggered during the measurements. However, aside from the WCET analysis it is useful for constructing execution time profiles because the average case is also covered.

Nowotsch et al. [71] proposed an interference sensitive WCET analysis. There, the WCET approach is extended by an analysis of the applications' resource usage, and the interference delay is computed. It is complemented by a runtime resource usage enforcement in order to ensure that the assumptions made during the analysis are met. This approach enables an independent analysis of the different applications executed on the multicore. However, the approach uses a model based on measurements of the worst-case memory access latencies. Thus, it faces the same issues as the hybrid WCET analysis. There are other similar

approaches in literature that aim to optimize the WCET overestimation (e.g. Kelter et al. [46, 45]) by applying more precise models of multicore processors. However, due to the lack of knowledge of the internal behavior of COTS components, the models might not represent the actual worst-case.

There are several approaches based on probabilistic timing analysis. An overview is given by Abella et al. [2]. These approaches aim for a valid timing analysis without requiring a detailed knowledge of the processor architecture. An example is presented by Cucu-Grosjean et al. [15]. There, the analysis is based on measurements conducted on the multicore processor and extreme value theory. The authors claim that only a low number of measurement runs leads to a trustworthy WCET analysis. However, probabilistic methods require that the software and hardware show probabilistic behavior [65]. A prove for this has not yet been shown for COTS components.

3.3. Scheduling and Mapping

Several approaches were published on either limiting the interferences by a specific schedule or upper bounding the interferences and dynamically schedule applications at runtime. In the following, execution models and adaptive scheduling, also for mixed criticality applications, are discussed.

3.3.1. Execution Model

A well-known pattern is the decoupling of interference-prone communication phases (accesses to memory) from computation-based (accesses only to local cache or scratch pad) execution phases. An example approach was published by Durrieu et al. [21, 32]. There, the AER execution model (A (acquisition), E (execution), and R (restitution)) is applied to the software running on the different cores. A static schedule ensures that, at any time, only one core executes software in either the acquisition or restitution phase.

The PREM (PRedictable Execution Model) presented by Pellizzoni et al. [76, 6] is a similar execution model. The authors distinguish between memory phases (access of cores to the memory), execution phases and I/O flow (e.g. DMA). In the schedule, execution phases can be either parallel to memory phases or I/O flow.

In both approaches, the applications are effectively isolated, but parallelism is restricted to execution sections. Furthermore, since the execution model has to be integrated in the software, this approach is not applicable to legacy software.

3.3.2. Adaptive Scheduling

A mixed criticality scheduling approach was presented by Fisher [26]. The schedule is time partitioned in critical and non-critical time slots. Critical applications run standalone while

non-critical applications can run in parallel. Thus, during the execution of the critical task, there are no interferences from the low-critical task. However, this leads to a very low utilization of the multicore processor.

Agrawal et al. [4] presented a scheduling approach utilizing dynamic memory bandwidth isolation. There, the computation of execution times and scheduling is performed in a common analysis. This is advantageous in the case of systems with different memory-intensive partitions. During runtime, budgets for processing time and memory bandwidth are enforced.

The Budget-Based RunTime Engine (BB-RTE) published by Girbal and Le Rhun [33] aims at safely scheduling high-critical legacy applications by temporarily suspending low-critical applications. A budget is computed for each shared hardware resource in terms of extra accesses such that the critical tasks can perform before their runtime is significantly impacted. When the budget is exceeded the low-critical tasks are suspended.

All the discussed approaches in this section target the execution of unmodified legacy applications. However, the multicore utilization is low, since either only one critical application is scheduled or due to the overestimation in the analysis of the budgets processor time is unused.

3.4. Contention-Free Hardware Design

Predictability by processor design is studied for example by Schoeberl et al. [86] where the Patmos, a time-predictable dual-issue microprocessor [87], is combined with a statically scheduled TDM time-predictable NoC. Goossens et al. [36] propose a NoC-Based multiprocessor architecture for mixed-time-criticality applications with different arbitration schemes.

The parMERASA project, presented by Ungerer et al. [94], aimed for a timing-analyzable multicore architecture for the execution of parallel real-time programs. This architecture is designed as clusters of cores connected by a dedicated NoC.

The predictability of the MESI (Modified, Exclusive, Shared, Invalid) cache coherence protocol was studied by Uhrig et al. [93]. Recommendations for a time predictable implementation are presented that completely removes the interferences and thus, allows for a WCET analysis.

The LEOPARD architecture, published by Hernández et al. [39], extends the LEON3 multicore processor to allow capturing the impact of jitter introduced by the on-chip resources. This allows for more precise WCET results.

All the presented approaches increase predictability on the cost of average performance and target real-time applications. From a WCET analysis perspective, these processors are the best solution for application in avionics. However, due to the small market of avionic products compared to consumer markets, the main chip vendors are not interested in selling such processor designs, and designing company specific chips has several drawbacks, such as

high costs and potential design errors.

3.5. Feedback Controllers

The use of feedback controllers in combination with real-time systems is not new. Examples are a closed loop controller developed by Maggio et al. [60], for dynamic resource allocation and power optimization of multicore processors. A closed loop control in a real-time scheduler is presented by Sahoo et al. [85] and Cucinotta et al. [14] while a controller for thermal control of a multicore processor is introduced by Fu et al. [31]. However, all of these methods require intrusive measurements and do not control the interferences between cores.

A feedback controller for tackling the interferences between cores in a mixed-criticality system was introduced by Kritikakou et al. [54]. The approach regularly checks if the critical tasks can tolerate the interferences due to other co-running tasks and suspends non-critical tasks to execute the high priority task in isolation. In order to perform regular checks, the critical tasks have to be instrumented with observation points. At these observation points, a runtime measurement is performed in order to determine the delay of the task due to interferences. This approach allows for a high utilization of the multicore processor and is very interesting for new applications; but because of the increased effort for re-certification due to instrumentation, it cannot be applied to legacy applications.

3.6. Fingerprinting

Fingerprinting in the context of this thesis is based on recording the performance counter events in discrete time steps. It is not to be confused with the fingerprinting presented by Smolens et al. [89]. There, a fingerprint is based on the history of internal processor states to generate a cryptographic signature.

A previous approach for characterizing an application's execution by the performance counters is presented by Duesterwald et al. [18]. It is used in high performance systems to predict an application's future behavior and needs for adjusting architectural parameters for performance optimizations. It is not related to embedded real-time systems but successfully uses a similar, but intrusive, technology for tracking an application's performance.

3.7. Summary

Even though contention free hardware is the best solution from a safety point of view, such processors are not available on the market and expensive in case of a company own design. In the following, the discussion will only focus on COTS components.

The limiting of interference is only achievable up to a certain extend. Robust partitioning is not achieved by any of the approaches. However, with upper bounding of the interferences

a WCET analyzable system can be created without a high overestimation. However, if the COTS component is not fully understood, the assumptions which have been taken based on measurements might not be correct. The same problem is faced by novel WCET measurement based approaches.

Execution models are very interesting for newly developed applications but not usable for legacy applications since the software has to be modified. The achieved utilization of the approach depends on the application. Runtime WCET controllers are a great approach for newly developed applications on a mixed criticality system; but as the software has to be instrumented, it is not applicable to legacy code.

The approach presented in this thesis aims on applicability to legacy applications while still achieving a high utilization of the COTS multicore processor. Similar to the idea of a runtime WCET controller, it shall be applied to mixed-criticality systems. Instead of instrumenting the code, the progress shall be determined non-intrusively by fingerprints of the application. Furthermore, the observation and control shall be performed by a device external to the multicore processor. None of the presented approaches utilize external devices even though in that case, the runtime-enforcement itself is not prone to interferences.

4. Safety-Net

The goal of this work is to provide means for the safe and efficient use of multicore processors in avionic systems while reusing existing applications without modification. The safety-net presented in the following focuses on the timing of a critical application, since the main issue with multicore processors is the timing unpredictability and analyzability.

The assumed setup where the safety-net shall be applied is a COTS multicore processor where one core is defined to be the critical core. On this core, applications with a high design assurance level (e.g. DAL-A) can be executed either directly on the core or as part of an IMA system together with other low and high critical applications. On the other cores, applications of lower criticality can be executed.

There are five main objectives for the approach.

1. The timing of the critical applications shall be kept. These applications have to finish in every period within a defined deadline. This is the most important goal.

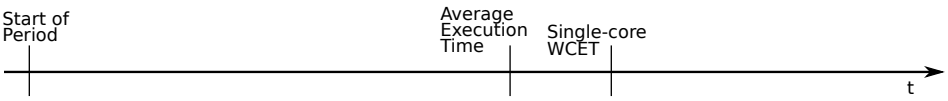
2. The processing resources of all the cores shall be used efficiently.

3. Existing single-core code shall be used on the multicore without modifying significant parts of the code of the application because changes in the code cause higher effort during recertification.

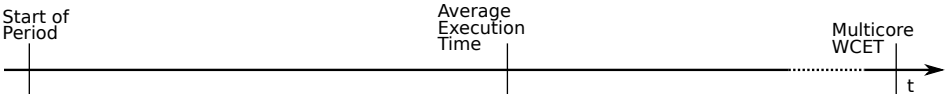
4. It shall be possible to exchange and certify applications without recertification of the other applications on the processor. Usually, if one application is changed on the multicore system, even if it is a low critical application, the timing analysis of the critical application has to be redone as it can be influenced by the new application via the interference channels. This shall not be necessary with the safety-net approach.

5. The last goal is non-intrusiveness. The timing isolation of the individual applications shall not slow down the execution of the programs.

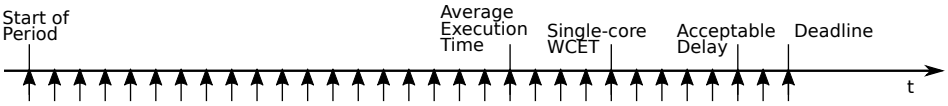
In this chapter, the main concepts of the timing isolation safety-net are covered. The basic idea and an overview of the approach is introduced in Section 4.1. In the following sections, the different building blocks of the approach are explained in detail as referenced in Figure 4.2. In Section 4.2 the prerequisites and demands on the multicore system and the running applications are described. The basic idea of fingerprinting, how to extract the fingerprints, and the creation of a model of the application is explained in sections 4.3, 4.4 and



(a) WCET approach for a single-core processor. The WCET is slightly higher than the average execution time.



(b) WCET approach for a multicore processor. Due to interferences the WCET is much higher than the average execution time (WCET overestimation) which leads to a low utilization of the processor.



(c) Safety-net approach on a multicore processor. The acceptable delay is a margin which is added to the single-core WCET. The tracking of the progress is performed in very small discrete time steps represented by the arrows.

Figure 4.1.: A comparison of the WCET and the safety-net approach on one period of a high critical application.

4.5. The detection of interferences using the model is presented in Section 4.6. Finally, the throttling of the interfering cores and the closed loop controller is introduced in sections 4.7 and 4.8.

4.1. Basic Idea

In contrast to the multicore worst-case execution time (WCET) approach, where the goal is to define a deadline that is going to be met even in case of interferences (figures 4.1a and b), in the safety-net approach a single-core WCET is determined which does not include the multicore typical high WCET overestimation (see Section 2.3) as shown in Figure 4.1b . This analysis is done standalone, the application(s) on the critical core are executed without any applications running on the other cores. This is legitimate as, during the actual execution with different realistic applications on the other cores, the runtime of a realistic application is similar to its standalone execution time most of the time. This is a result of the different levels of caching used on a COTS multicore. However, for a real-time DAL-A application, it has to be ensured that the deadlines of the critical application are always met. Meeting the deadlines in most of the cases is not sufficient and can lead to catastrophic events. Since it is possible that an application on one core creates high traffic on the interconnect and memory so that the single-core WCET is not met, the progress of the critical application has to be constantly monitored to ensure a correct timing. In order to slightly reduce the demands of very fast reaction times on the monitoring safety-net system and to account for some interference on the SoC, an acceptable delay is added. This is shown in Figure 4.1c.

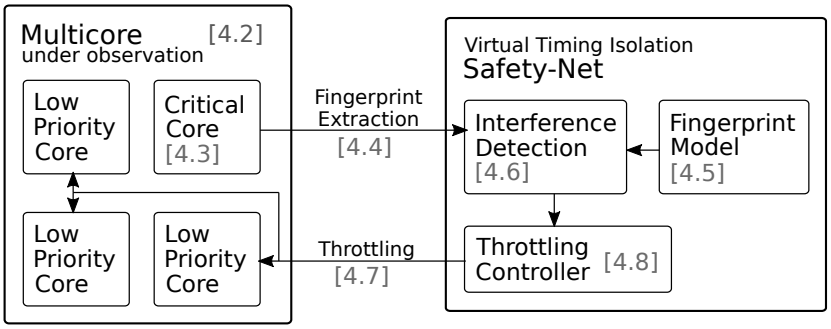


Figure 4.2.: Overview block diagram of the safety-net approach which shows the system on chip (SoC) under observation by the safety-net processor. The different building blocks are explained in the corresponding sections as denoted in brackets.

The acceptable delay is added on top of the single-core WCET and could be for example in the range of five to ten percent depending on the reaction time of the monitoring system and the expected interferences. Furthermore, it has to be placed shortly before the actual deadline. In order to determine the progress, the application has to be checked in very small time periods. One possible implementation is the instrumentation of the application with checkpoints where messages are sent to the monitoring system. However, this introduces the problem of higher effort for retesting and recertification as this involves a significant change of the program.

In the safety-net timing isolation approach, fingerprinting is used to perform these fine grain progress checks. This technique relies on the performance counters implemented in the cores, which does not need a change of the program or access to the source code. The application's progress is only tracked on the basis of characterized behavior of hardware event counters which are extracted from the critical core by an external safety-net system. This is shown in the overview in Figure 4.2 on the example of a multicore processor with four cores. Possible suitable performance counter events for the tracking are the number of executed instructions, cache misses, and executed floating-point operations based on a given time period (more detail in Section 4.3.2). Periodically reading and resetting such counters results in a curve that is characteristic for an executed application, more specifically, for the progress of that application. When comparing a recorded reference model, which is unique per application, to the performance counter values measured online, the current progress with respect to the reference execution can be measured (explained in Section 4.3). In case the performance of the critical application drops, the safety-net system is able to throttle the other cores to reduce interferences. An integrated controller is responsible for this task as shown in the figure. The reference model is created in standalone mode individually for every monitored application. This allows to exchange the applications on the different cores without the need for recertification of the complete system. As the timing of the critical application is virtually isolated from the interferences of the low critical applications, no common timing

analysis of all the software running on the different cores has to be done. The model creation and monitoring has to be done only for the critical application and is non-intrusive to the application. For compliance with CAST-32A the monitoring shall be done by an external processor which has to be a highly qualified (DAL-A) processor as certification relies on this. For this purpose, the safety-net processor and the software of the safety-net system is kept very simple.

4.2. Prerequisites / Environment

The approach is applicable to typical high critical avionic applications which are executed periodically, for example flight control or pilot assistance systems. It is assumed that the system is implemented as one critical core which hosts a single or multiple critical applications along with multiple low critical applications on the other cores. The monitoring of multiple critical cores of the same design assurance level is possible but not handled in this work.

There are no restrictions for the lower criticality cores. These applications can, in contrast to the critical core, also be executed in symmetric multiprocessing (SMP) (see Section 2.1.2) and IMA scheduling is not necessary. Furthermore, the code of the low critical applications does not have to be accessible and the applications can be exchanged without changing the model needed for the tracking.

This work focuses on the processing cores as the main sources of interference and neglects DMA and other on-chip devices that can create interferences. It is assumed that these devices are disabled.

4.3. Fingerprinting

During the execution of an application, a flow of instructions is executed. This flow is not homogeneous in terms of type of instructions (e.g. arithmetic, floating point, branch), source of the instructions (e.g. cache, internal scratchpad, external memory), and execution time of instructions (e.g. simple arithmetic, complex arithmetic, memory access). Accordingly, measuring for example the number of executed instructions per time unit will lead to a characteristic curve of an application or a part of the application. If the application is executed in an identical environment (e.g. processor, co-running applications) the measured curves are very similar. However, if the co-running applications change from a low-memory profile to a memory intensive application, the measured application suffers from interferences on the shared memory hierarchy, and its progress is slowed down. This results in a stretched (in time) but shrunk (in the value range) fingerprint (see Figure 4.3). The effect is more dominant at memory intensive parts of the code and less drastic at parts of the code which are fully cached.

When comparing such a mutated measured curve with the original reference curve, the progress of a known application can be tracked while the actual slowdown can not only be

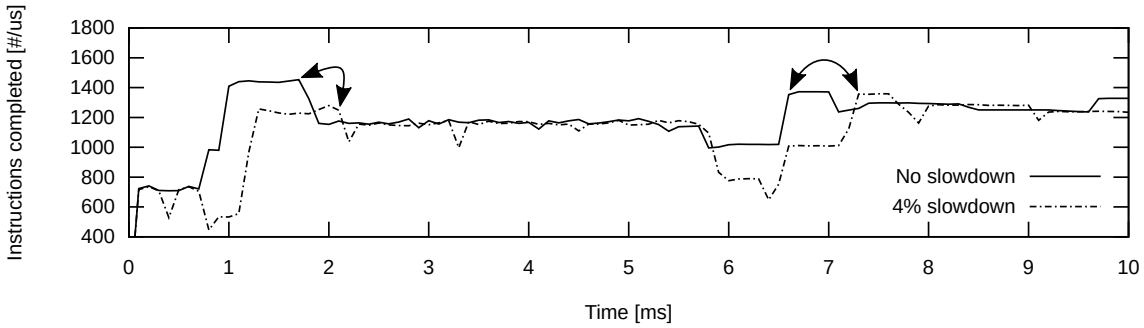


Figure 4.3.: Fingerprint curve with the event counter *Instructions Completed* of two executions of the TACLeBench benchmark suite. The solid line shows a run without co-running applications while the dashed line shows a 4 % delayed execution that suffers from interferences created by applications on the other cores. The arrows indicate the corresponding delayed execution.

identified but also be quantified at any time during execution. A similar fingerprinting was proposed in the past by Duesterwald et.al. [18] for predicting program behavior in high performance computing (see Section 3.6).

Many current MPSoC (e.g. based on ARM, PowerPC) include performance counters implemented in hardware that can be configured to increment every time a given event is raised. While the number of different events which can be configured is usually more than 100, the number of counters that can be incremented simultaneously is small (around four to six) [72]. Therefore, the events that are suitable for tracking have to be selected carefully.

Figure 4.4 presents an example fingerprint based on event counter curves of the TACLeBench [23] *sequential* benchmarks (more details in Section 6.1.1) for the four event types *Instructions completed*, *Instructions fetched*, *Branches completed*, and *Stores completed*. The displayed curves are recorded during a bare metal execution on an NXP P4080 with enabled L1 and L2 caches. The characteristics origin from the following algorithms within the TACLeBench in the following order: adpcm, anagram, audiobeam, cjpeg_transupp, cjpeg_wrbmp, epic, fmref, g723, gsm, h264, huff, ndes, petrinet, rijndael, statemate. These algorithms for example include jpeg transformations (7th to 12th ms), and AES decryption (32rd to 38th ms). In the figure, it is visible that the characteristics of the curves are different for the type of algorithms executed as well as the type of monitored events for the same algorithm.

4.3.1. Which Effects Generate Which Curves

There are two types of effects creating variations in the fingerprint curve: core intrinsic and core extrinsic. The intrinsic effects result from variations within core. These are, for example, the execution of different types of instructions. A sequence of ADD operations leads to a higher amount of instructions per second than a sequence of multiply operations or floating

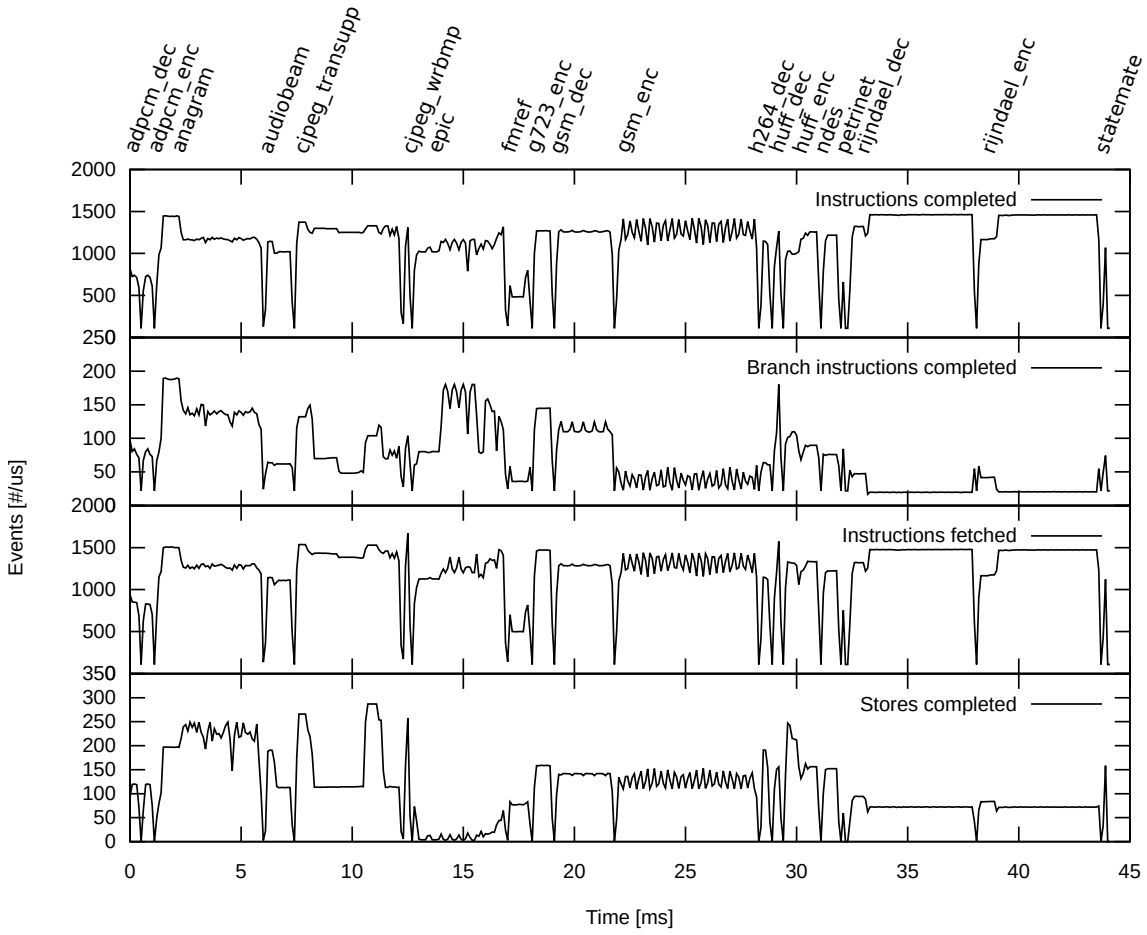


Figure 4.4.: Measured fingerprint curves of the four event counters with a sample period of 100 μs: *Instructions completed* on the top curve, *Instructions fetched* second, *Branches completed* third, and *Stores completed* in lowest curve when executing the sequential benchmarks of the TACLeBench [23] benchmark suite (Table A.1).

point operations.

An additional core intrinsic effect is branch miss-prediction. In order to achieve a higher throughput, modern processors rely on branch predictions. However, if the branch prediction algorithm has a low prediction success rate for a certain algorithm, the penalty results in a decreased throughput. This is also visible in Figure 4.4. The AES decryption (32rd to 38th ms) has a very high throughput, because, besides other aspects, there is only a very low number of branches in the code. Furthermore, the existing branches are very well predicted.

The third and major intrinsic influence on the fingerprinting curve is caching. If there is a high cache hit rate for a certain application, the throughput is obviously much higher because in this case, the processor does not have to wait for the much slower memory. In Figure 4.4 this is visible in the *audiobeam* benchmark where a huge amount of data has to be loaded and stored from/to memory which results in a lower amount of instructions completed per second. Especially interaction with on chip entities such as UART and other devices relevant in embedded computing takes a huge amount of cycles per instruction. These registers cannot be cached and can also not be read out of order. This appears in Figure 4.4 as drops of the curve to almost zero (e.g. at 6 ms), which is the result of a UART message after every benchmark. Furthermore, the cache configuration such as enable/disable of certain cache levels as well as cache coherence configuration influences the fingerprinting curve.

In contrast to intrinsic effects, the extrinsic effects are introduced by interferences with other cores or devices on the system-on-chip. Since interferences of one core to a co-running core can mainly happen at accesses to a common bus or interconnect, the major effect is limited to memory accesses. Other effects are explained in Section 2.2. Therefore, the more an application takes advantage of caching, the less it is prone to interferences.

An example for extrinsic effects is shown in Figure 4.3. Both curves result from the same program run in two different configurations. In the first configuration (*no slowdown*), the program was executed on one core of a multicore processor without co-running applications on the other cores. The other curve (4 % *slowdown*) shows the program run with co-running applications which have a very high memory access rate. In both cases, the number of instructions completed per second is displayed. Overall, the slowed down curve is similar to the original curve but stretched (4 % in total). However, the curve is not continuously stretched and dropped even though the co-running application accesses the memory continuously. There are sections which are very similar to the original curve. In these sections, the program is taking advantage of the caches and is not prone to suffer from interferences introduced by other devices on the SoC. The other sections show significant drops. This is especially visible at 1 ms, 7 ms and 21 ms. In these sections, the program performs many accesses to the memory and thus, suffers heavily from interferences which create the extrinsic features.

In the proposed safety-net approach, the intrinsic features are part of the reference fingerprint, while the extrinsic effects correlate with the amount of interference which shall be

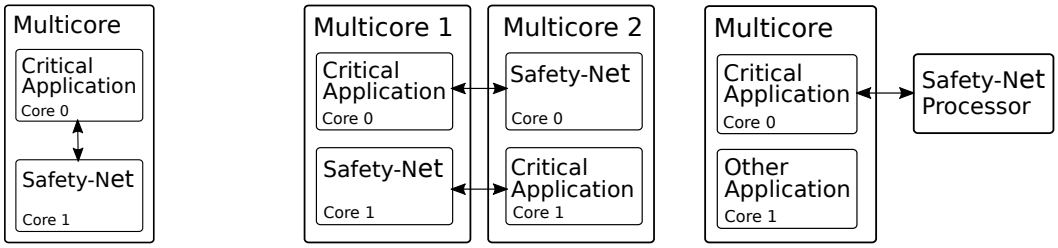
detected. Small blocks of instructions do not account for the shape of the curve as, depending on the sample rate and the frequency of the processor cores, more than 1000 instructions can lead to one data point in the fingerprint. Therefore, small variations within the code are not visible in the fingerprint.

4.3.2. Selection of Suitable Performance Counter Events

A fingerprint curve that can be used for progress monitoring has to fulfill the following requirements. First of all, it has to be a continuous stream of events per second. Sections with zero events per second longer than the sampling period cannot be used for the proposed approach. In this case, the tracking algorithm has no data to compare with previous recordings. Furthermore, a reduction of the measured events correlating to interferences cannot be detected. Second, for the best tracking precision, the stream of events should be varying over time. It is possible to track the progress for monotone sections in a curve since interference is also visible by lowering the height of a curve. However, different paths within a program are harder to detect.

For the fingerprint it is not relevant which specific performance counter event types are selected as long as it fulfills the aforementioned requirements. Furthermore, current processors provide counter registers for at least four different performance counter events. Therefore, it is sufficient if the combination of different performance counters fulfills the requirements. In order to allow for a maximum of robustness in the tracking algorithm, the selected event types result in different curves. In [72], the complete list of 180 possible performance counter events in the NXP e500mc core are given. However, in the following only the relevant events and groups of event types are presented in order to create usable fingerprint curves.

The first event type group are the *instruction types completed* events. For this group, the performance counter is incremented when the specific type of instruction (add, branch, load, store, etc.) finished in the pipeline. Additionally, there is an event for *Instructions completed* which represents any finished instruction. This specific event type creates a continuous flow of events and directly represents the progress of an application. Thus, it is selected as one of the four possible events. Furthermore, *Branch instructions completed* is selected as it also creates a continuous stream. It complements the *Instructions completed* because the number of branches per executed instruction is very application specific. Another group of event types are the memory and cache related events. These are for example *L2 cache accesses*, *Data L1 cache misses*, etc. The most relevant event for fingerprinting in this group is the *Bus Interface Unit (BIU) accesses* which are the L2 cache misses. Whenever this event occurs, the interconnect is accessed either for the purpose of memory accesses, or interaction with any other device on the system-on-chip. This is the main source of interference and thus, the event directly indicates possible interference locations in the progress of an application. In the co-processor event group, the *FPU instruction completed* is usable depending on whether



(a) Safety-net on the same SoC as the application.

(b) Dual redundant processors including safety-nets.

(c) Dedicated safety-net processor.

Figure 4.5.: Different possible safety-net architectures.

the application does include floating point operations. It does not necessarily produce a continuous flow of events, but it can be advantageous in identifying the correct path in a program. Especially, in cases where the stream of events is not varying over time for *Instructions completed*.

4.4. Extraction of Performance Counter Values

The performance counters are located in the cores of the application that shall be monitored. Thus, there have to be means to extract the performance counter values periodically. In the following, the different possibilities on how to read the values are discussed.

4.4.1. Possible Safety-Net Architectures

The options for reading the values from the cores depend on the location of the fingerprint processing node. This can be located within the multicore processor on a different core, or outside on a second processor. One possible architecture is shown in Figure 4.5a. In this case, the safety-net logic is executed on one core of the multicore processor that executes the application under observation on a different core. No external hardware is necessary and the sampling rate (the rate in which the performance counters are accessed) can be very high. This leads to a very precise fingerprint. Additionally, the readout procedure is very simple as the performance counter registers are usually memory mapped registers accessible from all cores within a multicore processor. However, the safety-net would be part of the unsafe system with all the drawbacks described in Section 2.4. If there is an error on the system-on-chip, not only the application fails but also the safety-net. One solution to this is two multicore processors observing each other's applications, as presented in Figure 4.5b. This is already a common approach in avionics where systems are designed redundantly and observe each other [90]. In contrast to the single processor solution, one processor can fail, and it would be detected by the safety-net running on the second processor. However, for both approaches the safety-net logic is executed on the processor which creates additional

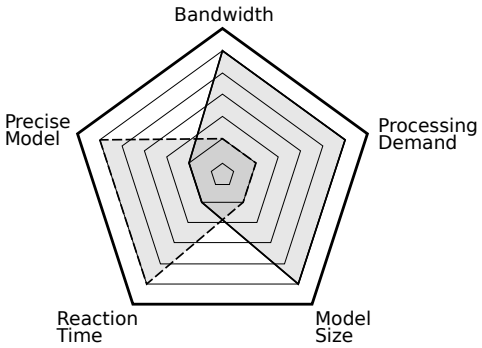


Figure 4.6.: Factors depending on the sample rate. The solid line represents a low sample rate while the dashed line shows a high sample rate. The further the line is at the edge, the more beneficial for the corresponding property. For example, a low sample rate is beneficial for bandwidth requirements, processing demand and the model size, but it leads to a less precise model and a longer reaction time.

interference for the observed application. Furthermore, the safety-net itself might suffer from interferences, which could lead to a delayed detection of slowdown of the observed application. This is one of the reasons why it also violates the recommendation of placing the safety-net external to the observed processor, as stated in CAST-32A (see Section 2.4).

The safest solution is a dedicated external safety-net as shown in Figure 4.5c. This safety-net processor must be a simple single-core processor, that can be certified to the highest certification level (DAL-A for avionics). This configuration is non-intrusive to the application under observation. It does not interfere with the application, and the application code does not have to be modified. Multiple applications on different multicore processors could be observed by one safety-net processor depending on the speed of the safety-net processor and the complexity of the observed applications. However, high-speed data transmission from the system under observation to the safety-net is required to read out the fingerprint data during runtime. Depending on the latency demands which correlate with the acceptable delay, this can be relaxed by pre-computing on the cores. The fingerprint data could be prepared on the multicore and only the relevant and compressed data is sent to the safety-net processor.

The remainder of the thesis will focus only on the dedicated safety-net architecture as it is the most promising towards certifiability.

4.4.2. Sample Rate

The sample rate is the frequency of the extraction of the performance counter values. The choice of the sample rate is not a simple task as it is a trade-off between five factors as displayed in Figure 4.6. A high sample rate ($< 10\,\mu$s period) leads to a precise model and a short reaction time. On the downside, it leads to a big model size which needs a high amount of storage space and might not fit into the attached memory but has to be off loaded

to external storage. Furthermore, the need for bandwidth between the SoC and the safety-net processor for the transfer of the PMC values is very high. For a dual-issue out of order processor, a maximum of two events per cycles can occur. Thus, the effective performance counter bit size required can be calculated as

$$bitsperpmc = \lceil log_2(2 * f_{cpu} * t_s) \rceil \tag{4.1}$$

with the CPU frequency f_{cpu} and the sample period t_s. For a core with a frequency of 1.5 GHz and a sample rate of 10 μs the maximum number of bits needed for a performance counter register is 15. This fits into the COTS registers which are typically 32 bits. The resulting needed bandwidth is then calculated as

$$B_{real} = \frac{n_c(n_{pmc}b + o)}{t_s} \tag{4.2}$$

with the number of cores n_c that shall be observed in parallel, the number of performance counters per core n_{pmc}, the size of a single performance counter register b, the transmission overhead for a packet of n_{pmc} values o and the sample period t_s. For a frequency of 1.5 GHz, a sample rate of 10 μs, 4 registers, 32 bits per register, an overhead of 200 % (more details in Section 5.4.3) and two observed cores, the needed bandwidth is 76.8 Mbit/s.

Additionally, the processing requirement of the safety-net processor is high because the tracking algorithm has to run in real-time. Thus, the safety-net processor has to read and process four data values at the speed of the sample rate. This performance requirement is especially important since a highly certifiable processor shall be used. In order for a processor to be certifiable, it has to be a very simple and thus, low performance CPU.

For a low sample rate ($> 1000\,\mu$s period), the opposite effects can be observed. The need for processing power, storage as well as bandwidth are lower while the precision of the model and the reaction time is going down. It depends on the observed application which compromise should be taken.

4.4.3. Extraction Interface

For the extraction of the performance counter values, a high-speed link from the multicore to the safety-net processor has to be established to satisfy the bandwidth requirements. Therefore, the options for such a link are Ethernet, PCIe or the debugging interface PowerPC Aurora / ARM HSSTP since UART and I2C are not suitable. Ethernet and PCIe provide the link speed but additional software has to be executed on the multicore processor to extract the data and send it to the respective interface. This introduces additional interferences on the system and requires the implementation of a task that reads the values and sends it to the interface. That task has to be integrated into the schedule of the multicore processor in very short time steps. In contrast to this, the debug unit allows for a non-intrusive extraction of

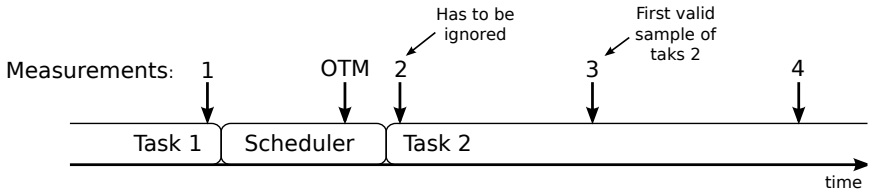


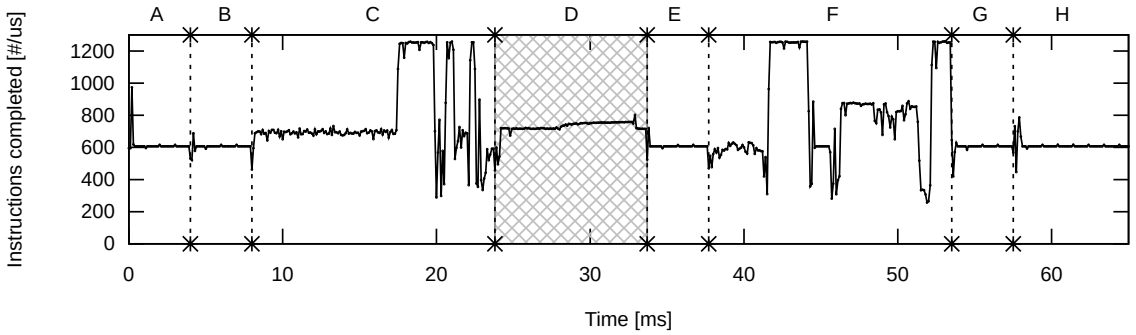
Figure 4.7.: Partition or task switches and the respective ownership trace messages (OTM) are not aligned with the measurement periods. Measurement 2, which is directly following the task/partition switch has to be ignored because it is not certain which portion of the measurement has to be assigned to task 1, task 2 and the scheduler.

the performance counter values without executing any additional code on the system under observation. The debug unit can be configured from outside so that no initialization by an operating system has to be performed. The bandwidth of such a tracing port is typically more than one Gbit per second which is more than sufficient for the proposed approach. There is one drawback for the use of the debug interface. Usually, the debug facilities on are not declared to be as reliable as the non-debugging features of the SoC by the system vendor; thus, it is recommended that it should not be used during flight. However, there is ongoing research proposing to use the debug unit in safety-critical environments (e.g. [34], [16] and [13]).

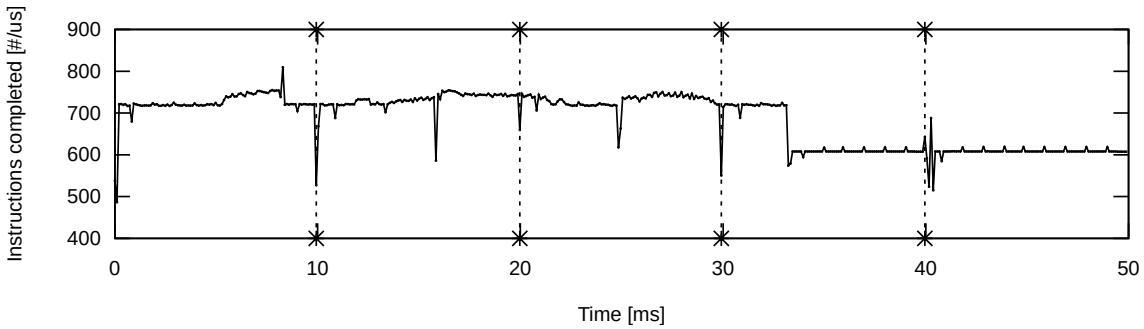
In the case of the fingerprinting proposed in this thesis, the whole COTS system-on-chip is expected to be unreliable since not all the documentation is disclosed to the aircraft vendors; therefore, not all the details are known as explained in Section 2.4. Additionally, a failure of the debug unit would be detected instantly when the measured curves do not fit to the prerecorded fingerprint. The only case which would compromise the safety-net system is the creation of model fitting measurements by the debug unit, while the actual performance counters indicate interferences (false-negative). It is highly unlikely that the debug unit creates such a curve as a result of an error. In the following, this thesis will focus on the debugging unit as extraction port because of the aforementioned advantages over the other interfaces.

4.4.4. Task Switches

The performance counters within a core count events on the lowest level possible. Thus, these counters do not distinguish between operating system code or the execution of different tasks. For the proposed approach, it is necessary to create a fingerprint model separately for every task. Differentiation between performance counter measurement values of different tasks is crucial. This can be accomplished by two means. The first solution is a message to the safety-net processor carrying the task id sent by the scheduler whenever a new task is scheduled. Even though this is an interesting solution for newly developed applications, it is not suitable



(a) Fingerprint curve of a major cycle of a real DAL-C IMA application which consists of eight partitions (A to H). The recorded event is *instructions completed*.



(b) Concatenation of the control partition (D) extracted from five subsequent major cycles.

Figure 4.8.: Splitting and concatenation of fingerprints resulting from IMA partitions.

for legacy applications as for this approach, the scheduler has to be modified, which is not wanted because of recertification aspects (more details in Section 4.2). The second option is the use of the debugging unit for this task. In modern processors, every core is equipped with a process id register to which the current task id is written to by the scheduler. The debug unit can be configured to send a message to the trace port whenever this register is modified. For the PowerPC architectures, this is called Ownership Trace Message (OTM). It provides a non-intrusive way of extracting the task id information and the application code does not have to be modified. In order to split the stream of measurements into chunks corresponding to the tasks, the first measurement of every task has to be withdrawn as it is uncertain how many events belong to the suspended task and the new task. This is shown in Figure 4.7. Measurement 1 is a valid measurement for task 1. Between measurement 1 and 2 the task switch occurs, and it is not known how many events belong to task 1, task 2 or the scheduler. The measurement must be rejected. Measurement 3 is the first valid value for task 2.

One special case in avionic systems is the *Integrated Modular Avionics (IMA)* architecture. Here, applications are separated in fixed time partitions (more detail in Section 2.1.2). An example of one major time frame is shown in Figure 4.8a where the different partitions are labeled as *A* to *H*. The main principle is identical to task switches. Whenever a partition

switch occurs, the partition id is written to the task id register and an ownership trace message is sent to the safety-net processor. Figure 4.8b shows one period of partition D that is concatenated of five chunks (five major cycles) of 10 ms each.

4.5. Model Creation

Branches inside of programs are taken based on the given input/sensor values. Thus, the fingerprint curve might be different for different input values. All the possible paths must be recorded in standalone mode to be able to track the progress afterwards. Therefore, a model that covers all the possible fingerprints has to be created. This model has to be encoded so that all possible paths are accessible in real-time during the tracking phase. The recording and encoding itself can be executed offline without timing constraints on a powerful compute node. Furthermore, the model has to be small enough for the memory of the safety-net processor. The creation of a fingerprint model is only possible for periodic applications like typical applications in avionics as explained in Section 4.2. These periodic programs have a defined starting point and execution time. Therefore, the model can be built as a tree, where the entry point of every repetition of the programs is the root. In this section, different approaches on how such a model can be created and encoded are discussed.

4.5.1. Recording All Possible Fingerprints

It is desirable for the fingerprint model to be complete. This can be achieved by executing the application several (thousand) times, without any co-running applications, while the performance counters are recorded. During the execution, all possible input parameters and internal states have to be set. Depending on the application, the possible combinations of parameters can be extremely high so that a complete model can only be achieved for very simple applications. However, this problem is similar for code coverage analyses where it is unlikely that 100 % test coverage is achieved (depending on the type of code coverage analysis). As such an analysis usually has to be performed for safety-critical applications (see Section 2.1.1), it is possible to combine it with the fingerprint recording. If during the tracking phase a measured fingerprint curve cannot be correlated to the recorded model, either a fault happened or the system is in an untested state.

The read-out frequency of the performance counter values of the selected events is recommended to be identical to the tracking frequency to allow for a simple and fast tracking.

4.5.2. Clustering

The recorded fingerprints result in slightly different curves even for identical input parameters because of local effects such as caching effects or influences by the operating system. This is visible in Figure 4.9 where an overlay of ten runs of a program with identical input parameters

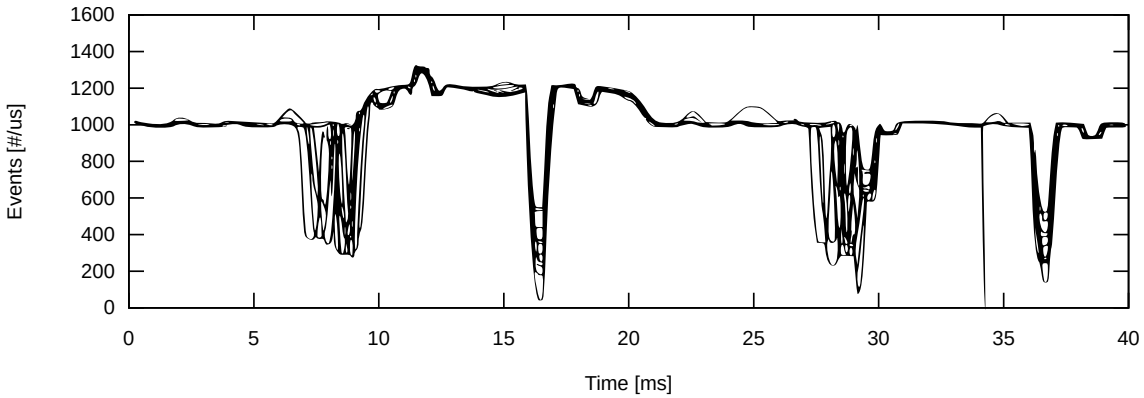


Figure 4.9.: Fingerprint recording for ten different runs with the same input parameters of a real avionics application.

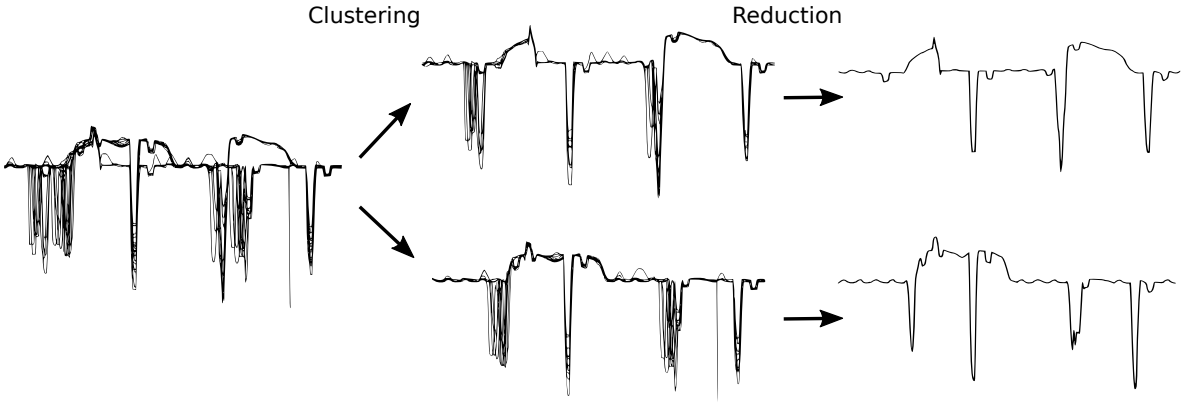


Figure 4.10.: Generation process of the Fingerprint model. The raw data (left) is clustered by a bisecting k-means algorithm and is reduced to the median curve to build up the Fingerprint model.

is shown. The overall shape is very similar, but especially at 8 ms and 28 ms the curves drop at different positions. In order to create a universal model, these similar curves have to be reduced to one curve. Furthermore, program paths that behave differently for very small parts only can be combined to one path in the model. This is possible depending on the size of these sections and the reaction time of the safety-net system. Applying that, the data to be stored in the model can be drastically reduced.

Depending on the observed application, the runtime of an application can be different for different input parameters. Clustering depending on the shape can only be performed on data sets with identical length. Thus, before the data can be clustered depending on the shape, a simple clustering has to be done based on the length of the individual execution. Here, the curves are sorted in buckets of identical sizes, e.g. 10 ms. Afterwards, for every bucket, a clustering can be performed based on the shape of the curves.

Clustering algorithms provide means to group similar data points. One standard clustering

approach is the *k-means* algorithm which was initially proposed in [59]. However, for this algorithm the number of resulting clusters has to be predefined. This is not possible for the separation of recorded fingerprints as it is not known beforehand, whether all the curves are similar or very different to each other. An extension of *k-means* that solves this problem is the *bisecting k-means* algorithm [38].

Algorithm 1: Bisecting k-means for clustering the fingerprint curves adapted from the proposal in [38].

```
function BisectingKMeans(cluster, centroid):
    Calculate distance d of the centroid to the curves in the cluster with Equation 4.4
    if (d > d_max) then
        Apply the k-means algorithm (Algorithm 2) to the cluster with k=2
        Call BisectingKMeans for resulting cluster 1
        Call BisectingKMeans for resulting cluster 2
    end
    return
```

This algorithm splits the number of curves recursively into two clusters until an exit condition is reached (see Algorithm 1). The centroid is defined randomly for the first iteration of the algorithm and refined in the subsequent iterations until the clusters reach their final states. In case of the fingerprint curves, the exit condition is the maximum distance d_{max} of all curves to the centroid of the respective cluster. The assignment of the curves to two individual clusters is handled by the standard *k-means* algorithm [58] (see Algorithm 2). The result of the algorithm is a number of clusters which holds curves with a distance $d \leq d_{max}$ to their centroid. With this approach, clustering of a given input set of curves can be done as sketched in Figure 4.10.

Algorithm 2: K-means approach for splitting a set of fingerprint curves into two clusters.

```
while there is no curve that changes the cluster do
    forall curves in recorded curves do
        forall centroids do
            Calculate distance between the centroid and the curve with Equation 4.4
            Assign the curve to the fitting cluster
        end
    end
    forall clusters do
        Calculate the cluster mean (centroid)
        Assign the cluster to the centroid
    end
end
```

The default distance function for the *k-means* algorithm is the sum of squared errors:

$$d(\mathbf{x}, \mathbf{c}) = \sum_{i=1}^{n} (x_i - c_i)^2 \tag{4.3}$$

with fingerprint vector $\mathbf{x}$, centroid vector $\mathbf{c}$ and the numbers of samples in the curve n. However, this function takes every error into account. For example, two curves in Figure 4.10 might not be in the same cluster because the drops at the beginning of the curve are slightly shifted even though the main stream of both curves fits perfectly. Therefore, a different distance function was chosen which is not that influenced by local variations but by the overall stream. The designed distance function is [1]

$$d(\mathbf{x}, \mathbf{c}) = \frac{\sum_{i=1}^{n} [|x_i - c_i| > \beta_{max}]}{n} \tag{4.4}$$

with runtime measurement vector $\mathbf{x}$, centroid vector $\mathbf{c}$, length n of the pattern and the maximum difference between two data points β_{max}. It does not sum up the differences between each measurement point, but it adds the number of samples per curve within a hull around the centroid. It sums up the samples with an error higher than the given β_{max} compared to the centroid. In comparison to the standard distance function, this function is not sensitive to drops in the curves. Errors bigger than β_{max} are taken equally into account which better clusters the main streams within the recorded data. The distance is divided by the number of samples per curve in order to be able to provide the same maximum distance d_{max} for all cluster sizes (buckets) as exit condition in the *bisecting k-means*. The maximum difference between two data points β_{max} is to be defined individually per performance counter, sample rate, and processing speed of the observed CPU.

This clustering has to be done for the recordings of all the different performance counter values. It is possible that for different input parameters a curve in performance counter A results in identical curves. However, for performance counter B it results in different curves. As a result of this, a matching has to be maintained between the different PMC curves. A simple approach for this is an iterative clustering where in a first step the values for performance counter A are clustered. Afterwards, the PMC B values are assigned to the same clusters as A. Sub-clusters are created if the distance of the curves of B is higher than d_{max}. In this case, the cluster of A is split in order to keep the relationship between the different performance counter values. This procedure is then subsequently performed for all the different performance counters.

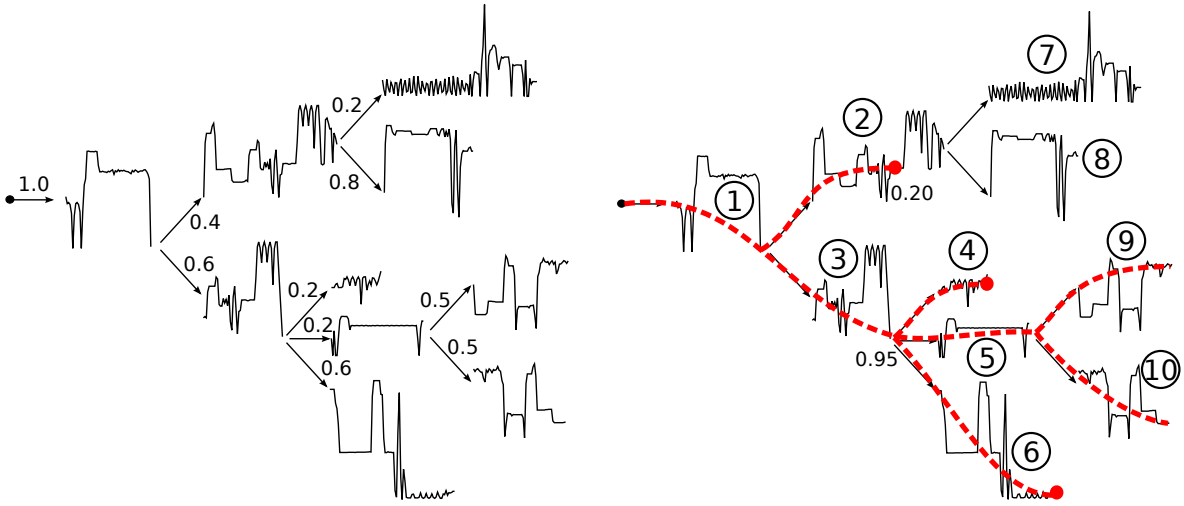
[1] Please note the Iverson brackets: $[P] = \begin{cases} 1 & \text{if } P \text{ is true;} \\ 0 & \text{otherwise.} \end{cases}$

4.5.3. Model Encoding

To further filter deviations from the main stream, for every cluster the median of the values is combined into the model. In contrast to the mean of the values, the median ignores deep drops which might be shifted in the curves. In Figure 4.10 it is visible that the drops at the beginning of the curve are reduced compared to the different curves in the cluster. Applying the mean values of the curves, deep drops have a much higher influence on the resulting curve.

During the tracking phase, an algorithm has to decide which of the recorded curves fit to the measured data. As this has to be done in real-time, not all the curves can be compared simultaneously. One solution to this is the creation of the model as a tree. This way, the decision which of the stored curves is the final one is split in several decisions over time. A sketch of such a model is presented in Figure 4.11a. A node describes a sequence of counter values, characterizing a part of the application's execution. The edges are links to the possible follow-on curves. The root node is the starting point of a new cycle of the application. One node consists of at least the number of simultaneously compared samples during tracking (referred to as δ in Section 4.6.1). Depending on the application, one node can include all the samples until the end of the period of the application or the next branch of the tree. For example, if during the tracking 4 samples are compared at any time ($\delta := 4$), the minimum size of a node in the model is 4. Each node has at least one successor node, except for the last node representing the end of execution in one iteration. In case of multiple successor nodes, they represent (at least) two different paths of following characteristics. The split into multiple paths can be caused by different execution paths of the application or by different environmental conditions. For example, the first execution can have a different path than the following executions due to cold caches and warmed-up caches. It is possible that different execution paths are not explicitly visible in the *Fingerprint* if these paths show similar characteristics as others. Furthermore, branches in the tree do not necessary correlate with the branches of the program. In order to improve the speed of the tracking algorithm, a Markov chain can be created by assigning probabilities to the edges. These probabilities can be derived during the testing phase. However, these probabilities are usually not representative, because the focus is on testing all possible execution paths rather than simulating an actual flight. Thus, more precise values can only be derived during an actual flight test.

The model presented in Figure 4.11a represents only one performance counter. For different performance counters, a model can be created independently or commonly per event counter. The advantage of independent models is the higher tracking robustness. If during the tracking phase model M_i is at a junction, model M_j might be still in a node and thus, reduce the juncture decision for M_i to a smaller amount of options. However, independent encoding is much more complicated to implement because in addition to the tree structures, a relationship between the nodes of the different models has to be maintained. In this thesis, the focus is on common models.



(a) Example of a fingerprint model for one particular performance counter event, implemented as Markov chains. The probability of the execution of the individual model branches is shown.

(b) Tracking of the application during runtime. In this example, up to three paths are tracked in parallel. If there are new branches in the chain (after 3), less matching branches (2) are discarded (matching score 0.20 vs 0.95).

Figure 4.11.: Example of a *Fingerprint* model implemented as a tree model during model creation and tracking.

The algorithm for creating a common model is shown in Algorithm 3. One of the performance counters is selected as master. The *Instruction completed* events are a likely choice, because a continuous stream of events is produced and many different events are combined in this counter. All the branches in the tree are based on the master. The other performance counters are split and assigned to the nodes according to the master. Hence, these counters provide a third dimension to the tree and allow for a more robust tracking. In the first step of the model creation, a random curve of the master PMC is selected to be the root node (chosen from the reduced data set in Figure 4.10). Afterwards, the algorithm loops over the set of curves and matches it to the tree starting at the root. If the new curve deviates from the model, the node in the model as well as the new curve is split. Edges are created pointing from the leading part of the node in the model to the trailing part of the original node and the trailing part of the new curve. This is done in iterations until all the curves are integrated into the model.

An additional advantage of a tree model is the compression of the data to be stored as redundant parts are removed. This can be further extended by encoding loops in the model. However, in order to be able to store models for applications with a huge number of paths, different models for different applications simultaneously or models with a very high sampling rate, a higher compression of the model is necessary. This could be done lossless by reducing the number of bits which are used to encode one performance counter value. This is possible as the maximum number of bits is known, depending on the type of core and the frequency

Algorithm 3: Model creation from the medians of the clusters.

```
Set random curve as root node to tree model and remove curve from set of median
 curves of the clusters
forall median curves of the clusters do
    forall data points in curve[# different events] do
        if distance from model is bigger d then
            Split current node in model in two nodes
            Link residuals of the curve to current node in model
        end
    end
end
```

(see Equation 4.1).

4.6. Interference Detection Algorithm

During the actual execution time, the safety-net processor shall compare the model to the measured performance counter values in real-time to detect a slowdown of the critical application. Other applications are co-running on the processor, either on different cores or as IMA system on the same core. In this section, approaches for tracking the progress of an application considering different matching algorithms are presented. Furthermore, means for interference identification are given.

4.6.1. Tracking the Progress of Applications in Real-Time

Similar to the process of model creation, the performance counters are read from the observed processor as described in Section 4.4. In contrast to model creation, the values cannot be stored and processed off-line because the tracking analysis has to be performed in real-time according to the reaction time constraints of the safety-net system. Furthermore, processing has to be done with the limited performance of a microcontroller for two reasons. First of all, it does economically not make sense to observe the multicore with a high-power single-core processor or even a multicore processor. Secondly, the safety-net processor has to be highly certifiable as it has a direct influence on the running applications, and thus, only a very simple single-core processor is applicable (see Section 4.4.1).

The read-out frequency in general can deviate from the read-out frequency during the model creation process. However, a higher sample rate during tracking leads to a higher demand on processing power and bandwidth, but no gain can be achieved from this. A higher frequency during recording allows for the creation of a more precise model while saving processing power during progress monitoring. This can be improved by applying a higher sample rate at task switches to align the measurement data to the model at the root node.

The tracking of every application starts at the root node. Thus, the safety-net has to be activated before the start of an application. Another option is the use of a debug message to announce the start of a new iteration of an application. Pattern recognition for detecting the start is also possible.

If the starting point is aligned, the measurements are compared to the values of the current node in the tree. The comparison has to be done to different parts of the node, the current section and a shifted version, in order to detect a slowdown and keep the measurements aligned to the model. Since measurements and *Fingerprint* never match exactly because of different execution environments (cache state, concurrent bus and memory accesses) or just because of jitter at the measuring points, the tracking algorithm is based on similarities not on exact equality. This is of special importance at the edges of the *Fingerprint* model because here the algorithm has to decide which path to continue and which node fits to the current execution and which introduces an uncertainty. This principle is sketched in Figure 4.11b. Several possible nodes have to be matched to the measurements. If the tree is encoded as a Markov chain the probabilities of the edges support the decision process. The most probable paths are selected first, which speeds up the decision. In order to increase the robustness of the algorithm despite of the uncertainty, multiple paths are tracked in parallel. In the figure, a maximum of three different paths are tracked simultaneously. At the decision at node three, the path following node two is terminated because the correlation of the measurements for the execution run up to this point was higher for node three than node two (0.95 compared to 0.20). The nodes following node three are terminated at the next branch. This way, even if there is an extrinsic feature in the measured curve directly at the beginning of a node, the algorithm is able to correlate the measurements to the correct path.

The algorithm can lose the track for different reasons. First of all, it might be that the path is not in the model because it was not recorded with the specific combination of input parameters. This should be prevented by the fingerprint recording methodology but can happen as it is not for all applications possible to record all the curves. Secondly, the program behaves incorrectly because of a programming error or because the processor suffers from a fault or single event upset. In this case, other safety-net functions that monitor the state of the processor recognize the fault. For programming errors, the dissimilar redundant unit (see Section 2.5) should detect the error. The third reason is an error that occurs during the matching of the measurements to the model. If there are extrinsic effects that shape the measured curve like a possible node during the decision process that lasts long enough to out-rule the true path, the tracking algorithm is lost, and no suitable matching can be found at the branch. This is very unlikely and can be identified by tracking the *Bus Unit Interface* performance counter as explained in Section 4.3.2. In case a measured curve is shaped like a wrong node in the model while at the same time the BIU accesses are high and interferences can be measured on the other cores, the error is detected.

In all the three cases, the algorithm detects that the track was lost. The safety-net is in an unsafe state but can assure the timely execution of the high-priority application by disabling the low priority applications until the tracking is recovered or, in the worst case, until the end of the current period of the program. At the next iteration, the safety-net tracking starts over at the root node of the model, and the low priority applications can be continued.

4.6.2. Matching

To track the current program execution path in the model based on the performance counter read-outs, the measurement samples have to be matched to the model. The expected result of the matching algorithm is the similarity of two patterns and the displacement of these patterns. The similarity indicates whether it is the correct branch in the model while the displacement can directly be converted into the slowdown of the application. This matching is a pattern recognition problem. However, it is constrained by the low performance of the safety-net processor and the real-time demand of the application in safety critical systems. In the following, different matching algorithms are explained, including a discussion on their applicability for the safety-net system.

Cross-correlation

One classical pattern recognition solution is offered by the *Cross-correlation* algorithm. It is used in signal processing to search for a small pattern in a longer signal. The result of the cross-correlation is the similarity and the displacement of a small feature in a longer series of data. The cross-correlation r for two series $x(i)$ and $y(i)$ for a displacement of d is defined in [78] as

$$r(d) = \frac{\sum_i [(x(i) - \overline{x}) \cdot (y(i-d) - \overline{y})]}{\sqrt{\sum_i (x(i) - \overline{x})^2} \sqrt{\sum_i (y(i-d) - \overline{y})^2}} \tag{4.5}$$

This algorithm can be applied to the safety-net approach to detect the features of the runtime measurements (y) in the model (series x). The result depending on d reveals the absolute similarity and the displacement. However, the cross-correlation works best for changes in values rather than absolute values. The absolute height of a curve is almost ignored but for the safety-net approach the absolute height is an important measure. The difference of the absolute height of the measured values compared to the recorded data is a direct measure for slowdown experienced by the application. This is especially the case in phases where the fingerprint curve is a relatively straight line. There, the cross-correlation indicates a high similarity even if the curves are unequally high. Even though cross-correlation algorithms can be implemented in FPGA logic, which can serve as a co-processor to the safety-net processor for speeding up the computation, the effort for comparing several patterns from the model to the real-time data is high. With the knowledge of previous correlations, the possible displacements can be reduced to a small number of data points and normalization

can be removed. Still, it is a computationally high effort. The algorithm has to be performed completely from the beginning for every new data sample because for example the average of the series is included in equation 4.5.

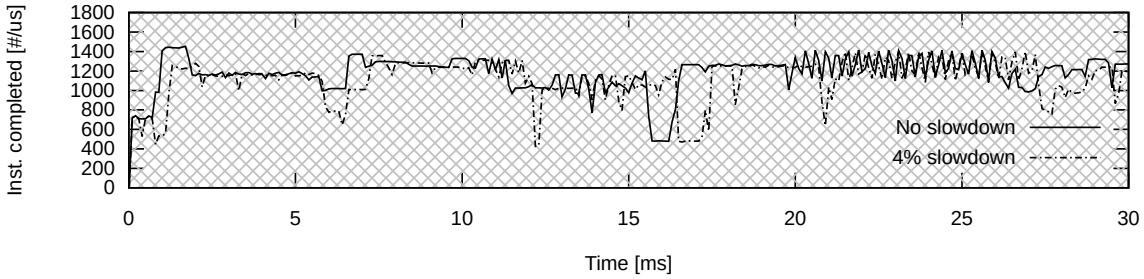
Euclidean Distance

A much simpler approach is the matching of two patterns using the Euclidean distance. In this case, the dissimilarity of two discrete time series is determined by the function

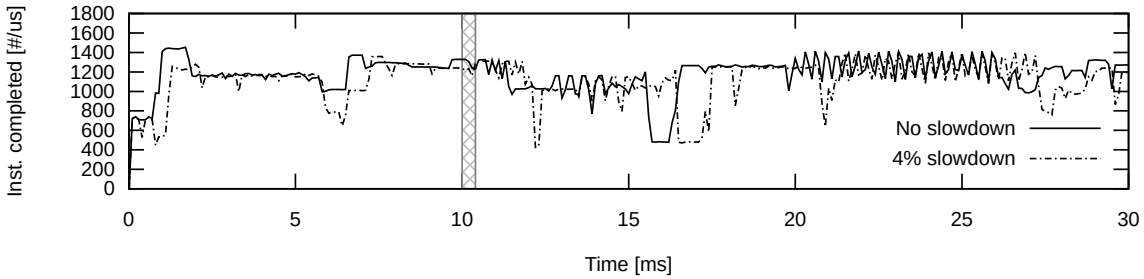
$$r(t,d) = \sum_{i=t-\delta}^{t} (x(i) - y(i-d))^2 \tag{4.6}$$

with runtime measurement vector $\mathbf{x}$, Fingerprint vector $\mathbf{y}$, discrete sampling time step from the beginning of the current period t, the number of discrete timesteps for comparison δ and the displacement d. This equation is applied directly to the raw fingerprint values every time a new measurement is taken. Therefore, after every reading, the most recent number of δ values is compared to the corresponding branches in the model. The result is the dissimilarity of the current measurements with the model. A lower r represents a low error between the functions and therefore shows as better fit of the curves. In order to detect extrinsic slowdown as described in Section 4.6.1, the equation has to be evaluated for shifted versions of the model which corresponds to the displacement d. This displacement can be not only greater or equal zero, which represents the case of slowdown, but it can also be negative to cover the unlikely event of a speedup of the program by extrinsic effects. These effects can be for example similar memory usage of different time partitions on the same processor. In this case, the cache could already be filled with the relevant data. From the result of r for different displacements, the local slowdown for the corresponding part of the program can be estimated. For example, if r results in a lower value for $d := 1$ than for $d := 0$ or $d := 2$, the slowdown is between 0.5 and 1.5 times the sample rate. In case the tracking algorithm detects several times the best fit for a certain displacement, the complete model is shifted to be aligned, the displacement is set to zero, and the local slowdown is added to the total slowdown of the current program iteration. Thus, the decision whether the complete application run fits to the *Fingerprint* is not only based on the most recent comparison but also on all the previous measurements.

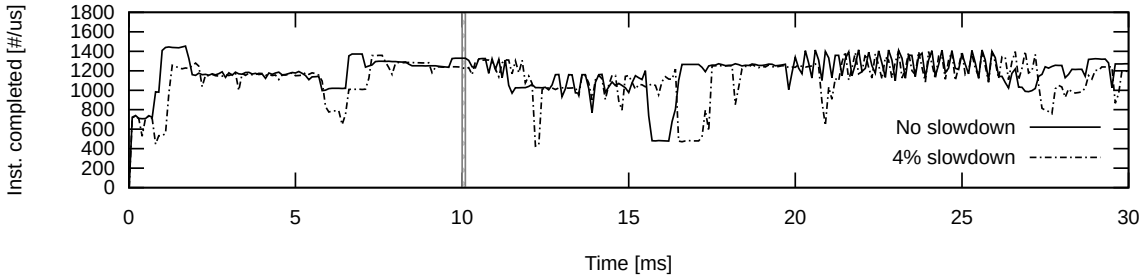
The selection of δ is crucial. If it is too high, a high-performance processor is needed for the processing in real-time. But even more importantly, in the extreme case of $\delta := \infty$ all the previous measurement values are compared to the model and a fit is only possible if there is no slowdown present (see Figure 4.12a). In this case, a fit is theoretically possible for an evenly stretched curve when such an artificial stretch is applied to the model. However, as described in Section 4.3.1, a fingerprint curve is only stretched evenly for a very specific and unrealistic combination of observed program and co-running application. Thus, a fit of big sections of the curve which includes parts that are stretched is not useful. For a low value



(a) All the measurement samples are compared to the model ($\delta := \infty$). Due to the stretching of the runtime curve, induced by interferences, a fit of both curves is not possible.



(b) Four measurement samples are simultaneously compared to the model ($\delta := 4$). A fit of four samples in the model and the measurement data is possible. In case a delay is detected, the curves can be aligned.



(c) One measurement sample is compared to the model in every comparison ($\delta := 1$). This provides a less robust tracking compared to four measurements because fluctuations in the curves may lead to wrong delay detections.

Figure 4.12.: Curve tracking for a different number of simultaneously compared samples (δ).

of δ (in the extreme case $\delta := 1$) the result is only based on single measurement values and does not take changes in the curve into account (see Figure 4.12c). Furthermore, results of a shifted model comparison only make sense for multiple measurement values. An example of a reasonable δ is shown in Figure 4.12b with a δ of four.

The computation performance required for this simple Euclidean distance approach is much lower compared to the cross-correlation approach. In every step, only the new measurement data has to be subtracted from the model data while the previous calculations can be reused. Afterwards the sum is computed in every step.

4.6.3. Interference Core Identification

In the previous sections, the approach for determining the progress and thus the slowdown of the critical application is explained. If a non-acceptable slowdown of a critical application is detected, the cores which create the interference have to reduce the interference in order to allow the critical application to finish in time. There are multiple approaches to select the core(s) to be throttled.

The simplest approach is to reduce the memory accesses of all cores except for the critical one. However, this also punishes cores that do not create a significant interference on the bus and the performance of the complete system is reduced even though it is not necessary. Another solution is to observe all the cores with the same methodology as the critical core. The low priority core with the highest slowdown is most probably the source of interference for the critical core and can be throttled. This would imply that fingerprint models have to be created for all the applications running on the different cores. This is a very laborious process and might not even be possible for some low critical applications which do not follow a defined IMA implementation with a fixed schedule as it is typical for critical applications.

Since both of the previously mentioned approaches are not satisfying, a new idea was developed. While observing only the progress of the critical applications with the fingerprinting approach, a lightweight observation is performed on the low criticality cores. This observation is, like the tracking approach, based on performance counters but instead of observing all PMCs available, only one is used. The corresponding event is configured to the *Bus Interface Unit BIU accesses* which is a direct indicator for the creation of interference on the shared infrastructure. For this observation, no model is created and thus, no tracking is done. It is only monitored how many times the interconnect is accessed by the low criticality core in a defined time period. In order to keep the overhead of the extraction of the event values low, the exact BIU performance counters are not accessed periodically, like in the fingerprinting approach. Instead, the performance counters are configured to send an overflow message to the safety-net processor once a certain threshold is reached. This does not create additional interference (more details are given in Section 5.4.2). The threshold is defined in such a way that, for an average application, the overflow is not reached for a long time span (e.g.

around 10 ms). However, for an application that creates a high traffic on the interconnect, the overflow is already reached below 1 ms. In the safety-net processor, the core that creates the highest frequency of BIU access overflows is expected to be the main source of interference, the cause of slowdown to the critical application and can be throttled first.

4.7. Throttling Techniques

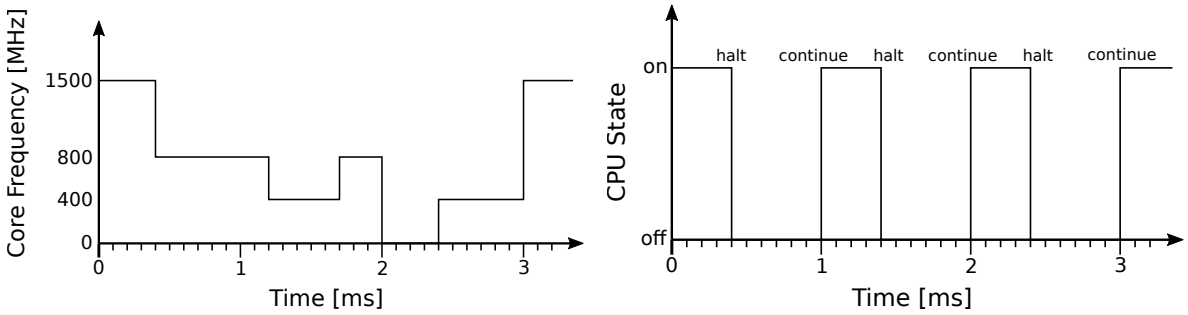
The *Fingerprinting* presented in the previous sections quantifies the real-time slowdown of the monitored applications. However, in order to keep the slowdown within the predefined boundaries of the acceptable delay, means have to be provided to lower the interferences introduced by the lower critical applications on the other cores. This can only be done by throttling the accesses of these cores to the commonly used interconnect and memory which results in a decreased performance for the throttled cores. Influencing the cores from the safety-net processor is only possible if a back channel is available provided by the debugging unit of the processor. The usage of the debug unit for influencing the cores has the same advantages as for the readout of the performance counters (see Section 4.4.3). The back-channel can either be implemented as a discrete line, which triggers an action within the debug unit, or as the return channel of a high-speed tracing interface. In order to take advantage of the full potential of the safety-net, the back-channel must provide a fast access and allow for controlling the cores individually. If no back channel is available, the interference cannot be controlled. In such a system it is only possible to switch to a backup system in case a critical application cannot meet its deadline.

Bandwidth Limitation

The ideal technique for throttling the cores that create interferences is a limitation of the bandwidth of the individual cores to the common interconnect. This is available for DMA blocks, but it is not available for the processing cores on systems like the NXP P4080 [74]. Thus, it cannot be used for current safety-net implementations.

Frequency Scaling

A reduction of memory accesses and thus interference of a low priority core can also be done by a frequency down-scaling. To which extend this can be applied in a COTS processor is highly dependent on the processor design. For example, the number of different clocks and the divider steps vary between processors. Furthermore, every core has to be configured individually since only the low priority core shall be slowed down. Many current processors support these features and the frequency scaling can even be configured via the debug interface. An example for the different configuration step options per core is given in Figure 4.13a. For a maximum primary frequency of 1.5 GHz a secondary clock frequency of 800 MHz can be selected. Both



(a) Different possible core frequencies on the example of the NXP P4080.

(b) Pulse width modulation like utilization of an individual core.

Figure 4.13.: Different throttling techniques applicable to modern multicore processors.

of these clocks can be divided by two, which leads to effectively three steps 1.5 GHz, 800 MHz and 400 MHz. This is configurable in very short time steps $< 100\,\mu$s individually per core. However, even though 400 MHz is small compared to 1.5 GHz the influence on the amount of memory accesses to the common interconnect is highly dependent on the software running on the core. If the software consists of many core-local operations between memory accesses, a frequency down-scaling has an effect on the amount of interference. In case almost only non-cached memory accesses are performed, the reduction of interference is not noticeable since the accesses to the comparably slow memory can only be performed at a lower frequency, even if the core speed is high. Therefore, frequency scaling can be used for a safety-net implementation only in combination with other throttling techniques as the throttling effect is highly dependent on the application.

Halt and Continue

The most simple and effective throttling approach is halt and continue of the individual cores. Multicore systems like the P4080 provide means to trigger the run states from the debug unit. Whenever a core is halted, the clocks are still running, but the core is not fetching or executing instructions [72]. Thus, no accesses to the memory are performed and the interference is stopped. Compared to the limitation of bandwidth and frequency scaling, this approach is very intrusive as the tasks on the respective core are completely stopped. This might have severe effects depending on the executed application.

A simple technique for reducing the interferences with the main application is halting and resuming the opponent cores based on a threshold. For example, whenever the slowdown of the main application rises over 5 %, the other cores are halted and continued once the slowdown drops under 5 %. Apart from the main goal of isolating the timing of the critical application, a sub goal is also to efficiently use the other cores. To provide not only a binary (on/off) way of setting the performance of the cores and allow the application the make progress, a (software-based) Pulse Width Modulated (PWM) enabling/disabling of the individual cores

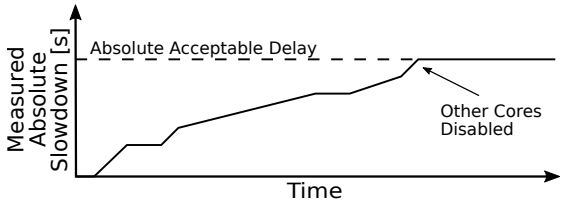
in short time periods is applied as shown in Figure 4.13b. In the example, the PWM period is 1 ms with a duty cycle of 40 %. The resulting active time span of a core is 0.4 ms within the 1 ms period. The halt and continue of a core are triggered by the safety-net processor directly with a register in the debug unit. This is performed on small time scales to provide a quick reaction time. If the measured slowdown changes during one period of the PWM, the halt or continue signal can be sent according to the new duty cycle. The PWM approach enables the safety-net to throttle the low critical applications in different levels.

4.8. Interference Controller

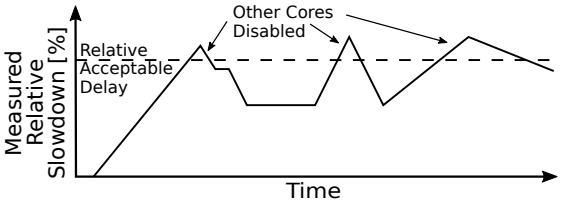
Interferences vary over time depending on the applications running on the SoC. Therefore, a closed control loop is established which reduces the interferences created by the low critical cores depending on the current slowdown of the critical application. This idea was published in [28]. In the control loop the sensor element is the measured slowdown of the observed application with the *Fingerprinting* approach and the BIU accesses of the low critical cores (see Section 4.6.3). The actuator which influences the performance of the other cores and, hence, the interferences, is represented by the frequency scaling or PWM throttling as explained in Section 4.7.

The primary goal of the controller is to keep the total delay of the critical application equal to or below the acceptable delay. In order to achieve this goal, the controller has to react very quickly as no overshoot is allowed. The worst-case interference is assumed to cause a complete stop of the application. Thus, the reaction time of the controller equals one sample period of the measurement algorithm. The secondary goal is a maximum progress of the low priority applications on the other cores and the maximum total utilization of the processor. Thus, the controller shall avoid throttling a low priority application if it is not necessary. Similar to the fingerprinting, a separate control loop has to be established for every IMA partition running on the critical core. In the following, possible controller designs are discussed.

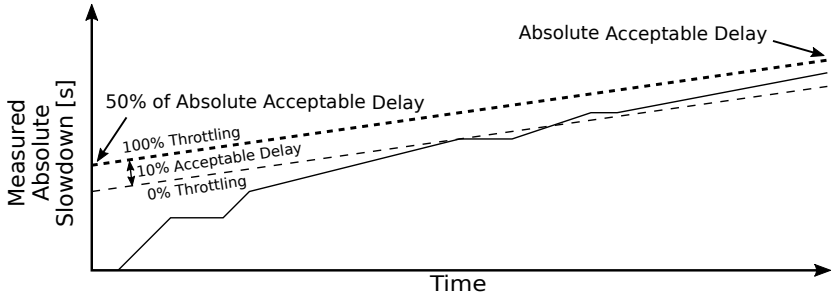
The simplest controller algorithm is a comparison of the absolute slowdown to the acceptable delay, which is shown in Figure 4.14a. If the measured delay reaches the acceptable delay, all the other cores are switched off. Even though this satisfies the primary goal, this is not ideal because all the applications are stopped even though it is possible that only one application creates the high interference on the shared resources, which violates the second goal. This is especially cumbersome in a system with multiple design assurance levels. It is not possible to only stop the core that creates the highest interference at that point in time because every program creates some interference. If the total maximum of acceptable delay is already reached, the risk of missing the deadline is too high.



(a) Algorithm based on a threshold of the absolute acceptable delay. If the total measured slowdown matches the acceptable delay the other cores are disabled.



(b) Algorithm based on a threshold of the relative acceptable delay. If the relative slowdown matches the acceptable delay the other cores are disabled.



(c) Combination of approach (a) and (b). The threshold is based on the absolute slowdown but dependent on time. A margin is introduced to gradually start throttling (0 to 100 %) the other cores, if the absolute slowdown is close to the acceptable delay threshold.

Figure 4.14.: Examples for different controller algorithms.

Another approach is to use a threshold with the relative delay

$$d_r = \frac{t_r + d_m}{t_r} - 1 \tag{4.7}$$

where the output of the fingerprinting measurement is the absolute delay d_m and the time from the beginning of the current loop t_r of the critical application. A resulting example curve is depicted in Figure 4.14b. If the current relative delay is higher than the slowdown, the low critical applications are stopped. This can also be implemented as a proportional controller where the amount of throttling is defined by how close the measured slowdown is to the acceptable delay. In this approach the reaction to slowdown is possible much earlier compared to the absolute delay approach, and the throttling focus can be on the most interference creating application. However, at the beginning of the run, the high critical program potentially needs a lot of data from memory before it is possible to take advantage of the cache. Hence, the cache warm-up is prone to create interferences. This initial delay will probably not be constant until the end of the period and the application might ultimately stay within the acceptable delay. In this case the secondary goal is not sufficiently met as low priority applications are throttled unnecessarily.

In order to solve the drawbacks of both controller approaches, a combination of both ideas was developed. An example curve is shown in Figure 4.14c. In this approach, the

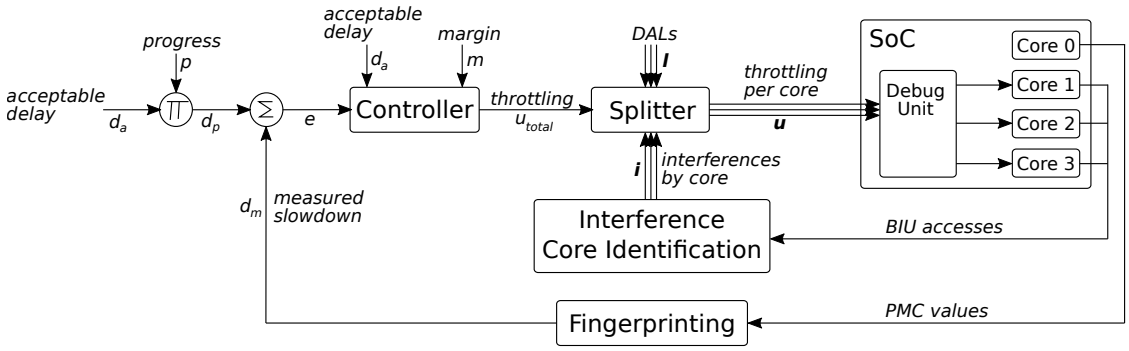


Figure 4.15.: Block diagram of the closed loop. The senor elements are the Fingerprinting for the critical application and the interference detection for the other cores in the feedback loop.

controlling parameter is the measured absolute delay d_m depending on the relative runtime of the algorithm, the progress

$$p = \frac{t_r}{t_{scWCET} + d_a} \tag{4.8}$$

with the time from the start of the current period of the monitored application t_r and the single-core WCET t_{scWCET} including the acceptable delay d_a (see Figure 4.1). This progress is saturated at 1 (100 % progress), in order to be able to use this variable after the acceptable delay is exceeded but before the deadline is reached. The threshold for this controller is a progress dependent absolute delay

$$d_p = \frac{d_a * p + d_a}{2} \tag{4.9}$$

with the acceptable delay d_a in s and the relative progress p. The threshold (upper dashed line in Figure 4.14c) is defined to start at 50 % of the acceptable delay and rises to 100 %. In this case, the absolute delay ensures high utilization of all the applications similar to the absolute approach. Additionally, counter measures are performed early enough similar to the relative approach. In contrast to the relative approach, which has the same effect as an absolute threshold from 0 % to 100 %, the combined approach allows for higher interferences at the start of the algorithm without throttling the low priority cores. A margin is introduced (lower dashed line) which acts like a conventional proportional controller to be able to start throttling only the interference causing cores. This margin can be defined; in the figure it is set to 10 %.

The block diagram for the corresponding control loop is shown in Figure 4.15. In this design, d_p, as defined in Equation 4.9 is calculated from the progress and the acceptable delay, is used as the setpoint for the control loop. The error e which is given to the controller is calculated as

$$e = d_p - d_m \tag{4.10}$$

with the variable threshold d_p and the measured delay d_m which is determined by the fingerprinting in the feedback loop. The controller function computes the amount of throttling

$$u_{total} = \frac{\frac{d_a}{m} - e}{\frac{d_a}{m}} \tag{4.11}$$

from the error e and the controller margin m. The output is normalized by the absolute acceptable delay d_a and saturated between 0 and 1. The margin is introduced to stop the controller from throttling the low priority cores if an irrelevantly small delay is experienced by the monitored application. With the margin, throttling only starts once a certain level of slowdown was reached.

As visible in Figure 4.15, the scalar output of the controller is the input to the splitter. This unit outputs the utilization of the low priority cores depending on the design assurance levels of the applications running on the respective cores and the amount of BIU accesses by the individual cores. The interference detection acts as sensor element in the feedback loop to detect the core which is most likely to cause the highest interference. The utilization vector

$$\mathbf{u}_{percore} = ((2 \cdot u_{total} - 1) \cdot \mathbf{1} + \mathbf{i}) \circ \mathbf{l}_{DAL} \tag{4.12}$$

is determined with the output of the interference detection algorithm as a vector $\mathbf{i}$ which holds the corresponding BIU accesses in the range from 0 (no accesses) to 1 (the maximum accesses compared to the other cores). The $\mathbf{l}_{DAL}$ vector holds the highest design assurance level of the applications running on the individual low priority cores. This has to be known beforehand and is statically configured in the safety-net processor. In this vector, the highest level (DAL-A) corresponds to 1.0 while the value is increased by η per level. Thus, the respective value for DAL-E is $1 + 4 \cdot \eta$. For example for $\eta := 0.05$ this leads to a values of 1.20 for DAL-E. Thus, depending on the design assurance level of the core, the level of throttling is increased by a maximum of 20 % in the example. Even though the main driver for the throttling is the interference detection, taking the DAL into account is especially beneficial if for example a DAL-C and DAL-E application create the same amount of bus interface unit accesses. In this case the DAL-E has lower utilization compared to the DAL-C application.

Equation 4.12 is designed so that for a total throttling of 1 all the cores are stopped even if the interference value for the particular core is 0. This is shown in Figure 4.16. For a u_{total} of 1 the resulting u is always greater than 1. Therefore, the throttling system is safe even in case of a malfunctioning interference core identification. Furthermore, cores are not stopped unnecessarily even if many bus unit accesses are detected ($i = 1$) but no slowdown of the critical application is observed ($u_{total} = 0$). In this case, the resulting u after the splitter is 0. For a total throttling of 0.5, the cores with interference value equal to 1 are completely switched off, while the cores with value 0 are still running without throttling.

The values of the splitter are saturated between 0 and 1 and forwarded to the actuator

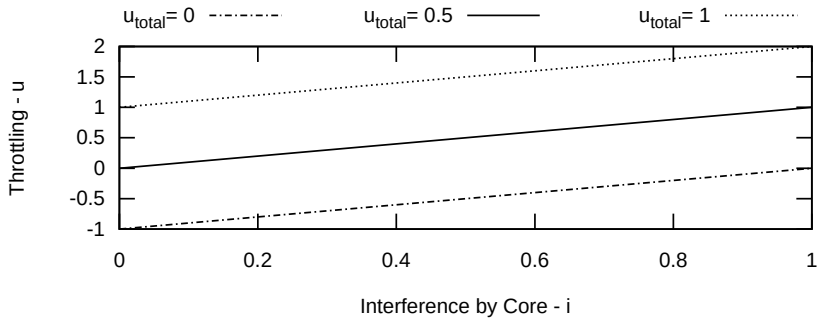


Figure 4.16.: The throttling u of a low priority core for a varying interference core detection i for different values of total throttling u_{total} according to Equation 4.12.

(frequency scaling/ PWM). Afterwards, the low priority cores are throttled via the debug unit. This closes the loop in Figure 4.15.

5. Implementation

In order to show the feasibility of a Fingerprinting safety-net, and its possible usage in avionics, the approach was implemented on the example of an NXP P4080 multicore processor. In the demonstrator system, a Xilinx FPGA is used for the timing isolation system which implements all functionality required for measuring and influencing the performance of the main application running on one core of the multicore processor. The FPGA reads the performance counters via the debug unit and any action to control the cores is also performed by this debug unit. The algorithm for model creation on a high-performance processor was implemented as well as the software for tracking the progress. The tracking algorithm is running in real-time on an embedded microcontroller.

In this chapter, an example implementation of a safety-net observing a multicore processor is presented. An overview of the hardware as well as a detailed description of the observed multicore and the safety-net processor including the interfaces is given in sections 5.1, 5.2 and 5.3. The software implementation of the extraction process is described in Section 5.4 followed by the model creation in Section 5.5. In Section 5.6 the implementation of the comparison of the recorded Fingerprint to the model is explained. Afterwards, the controller that uses the interference detection and triggers the throttling is presented in Section 5.7. Finally, an overview of the implemented methods for throttling the cores on an NXP P4080 is given in Section 5.8.

5.1. Hardware Overview

The hardware setup, consisting of a multicore processor under observation and the safety-net processor, is shown in Figure 5.1. The used multicore processor is an NXP P4080 with eight PowerPC cores placed on a P4080 development board revision v3. This board provides direct access to the debug interface via a 70 pin Samtec Aurora High-Speed Serial Trace Probe (HSTP) connector. On this connector, the bi-directional high-speed trace interface Aurora, JTAG and additional discrete lines are placed. Furthermore, apart from other devices, the board provides 1 GB of DDR3 memory and an RJ45 connection for Ethernet communication.

The safety-net processor is implemented as a soft-core Xilinx MicroBlaze inside of a Xilinx Virtex-7. The FPGA is placed on the evaluation board Xilinx VC707 which features two FMC connectors. These connectors are routed to individual pins of the FPGA (e.g. GPIO, SerDes Pins, ...) to allow for external expansion boards. A specifically developed FMC board

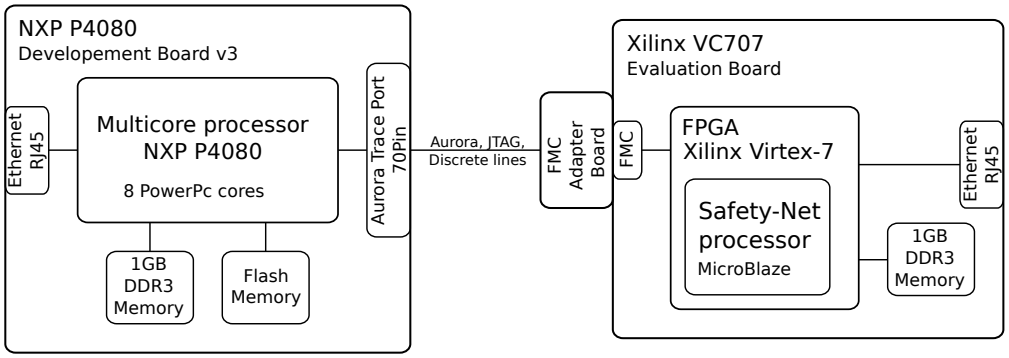


Figure 5.1.: Interfaces of the hardware setup used for implementation. The NXP P4080 multicore processor is connected to the timing isolation system implemented in a Xilinx Virtex 7 FPGA via a standardized Aurora trace port.

equipped with a Samtec Aurora connector is the interface to the multicore processor.

In Section 4.4.1 it is mentioned that the safety-net processor must be a highly-classifiable microcontroller. This is not the case for this FPGA. However, for the proof of concept, an FPGA was chosen because of the simple interface to the Aurora trace port of the multicore processor. The safety-net processor was replaced by a MicroBlaze soft-core which is comparable in terms of performance to high DAL systems. Thus, the performance restrictions for the real-time software apply equally on both implementations so that the developed software can be similarly applied to a certifiably hardware. In addition to the interface to Aurora, the implementation in an FPGA has several advantages during the development of a safety-net system. It allows for flexibility in the hardware setup and the usage of different co-processors, and pre-processing of the raw data sent by the multicore can be implemented quickly.

5.2. Observed Multicore Architecture

For the safety-net approach, the architecture of the observed multicore processor is irrelevant as long as the criteria defined in Section 4.4.3 are fulfilled. For the implementation in this thesis, the NXP P4080 was chosen because the PowerPC architecture is widely used in avionic applications.

The System-on-Chip (SoC) has eight 32 bit PowerPC e500mc cores. A block diagram of the SoC is shown in Figure 5.2. The maximum frequency is 1.5 GHz. The cores are equipped with a two-way superscalar pipeline with out-of-order execution. They contain a floating point, a branch prediction and a memory management unit. Every core has access to 32 kB instruction and 32 kB data level 1 cache. Additionally, a core local level 2 cache of 128 kB is implemented in every core. The access to the coherence fabric is provided via a bus interface unit (BIU). More details about the cores can be found in [72].

Apart from the cores, various other devices are implemented on the P4080 SoC. The

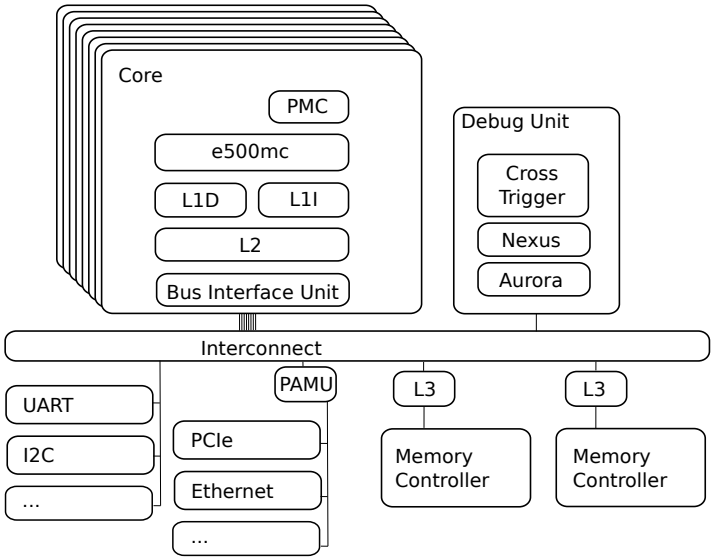


Figure 5.2.: Block diagram of the NXP P4080 [74]. The main sources of interference are the central interconnect and the memory controllers.

CoreNet Coherency Fabric maintains cache coherency in the different caches and is the central switch fabric for the communications between the cores, devices and memory. The platform is equipped with two 1 MB level 3 platform caches which can be configured as either cache or memory. The two DDR3 memory controllers which provide access to external memory are attached to these caches. There are many other devices, such as DMA, PCIe, Ethernet, etc. on the SoC, which potentially can access memory or even the caches of the individual cores. However, the access can be restricted by Peripheral Access Management Units (PAMU). There are several options for the boot-up procedure. In this thesis, the processor is configured to first load the bootloader stored on flash memory on the development board. Afterwards, the application to be observed is loaded via Ethernet from a TFTP server. More information on the system-on-chip can be found in [74].

This device is optimized for throughput rather than for predictability of real-time applications. It is very complex, and the documentation provided by the vendor is limited. Therefore, a complete analysis of the interference channels as mentioned in Section 2.2 is not possible. However, the main interference channels have been analyzed in [69], which resulted in a maximum worst-case execution time of 5.1 times the standalone execution time. This delay is highly dependent on the cache configuration and the executed application.

For the evaluation of the safety-net system at a real-life application (see Section 6.2.3), an NXP P5020 was observed. This processor has two e5500 cores which are the 64 bit equivalence of the e500mc cores used in the P4080. However, the SoC architecture is very similar. Especially, the debug interface is identical, which makes it possible to port the safety-net code to the P5020.

In contrast to microcontrollers, SoCs like the P4080 offer not only JTAG access to the individual cores. Instead, the debug unit is a complex on-chip device with a high amount of configuration and observation possibilities. The control and status registers of the debug unit are memory mapped and can be made accessible to the cores and the debug interface.

All the messages sent out of the processor by the debug unit are formatted according to the Nexus IEEE-ISTO 5001 standard [40]. The overall protocol is defined by the Nexus Forum, but vendor specific formats of each message type are possible. Therefore, the actual protocol definition for the P4080 can only be provided by NXP.

In the following, the individual debug unit blocks are explained including the debug assists in the individual cores.

Core Debug Facilities: In addition to standard features such as break points, every core offers a high amount of debug trace options, for example, program flow trace, data acquisition trace and ownership data trace (OTM). The OTM sends out a trace message to the SoC debug unit every time the thread id register in the core has been altered.

Every core is equipped with four 32 bit performance monitor counters (PMC) where each can be configured to count one out of 180 events. This is needed for the Fingerprinting as explained in Section 4.3.2. A complete list of the events that can be monitored can be found in [72]. The event counters can be copied to a PMC capture register to save the current state based on an event from the SoC debug unit. The cores also feature instruction jamming which allows to execute instructions written to a specific register. This is only possible if the core is in a halt state.

Run Control and Power Management: The RCPM has direct access to all the cores. It is able to change the run state of the cores (e.g. halt, doze, ...). Furthermore, the frequency of the individual cores can be modified. The RCPM can also trigger the debug facilities inside the cores, for example, to start and stop the performance counters [72]. All the actions performed by the RCPM are non-intrusive to the other cores since it is directly attached to the cores and does not use the CoreNet for communication.

Event Processing Unit: This unit provides counters and cross triggers for SoC level events such as Memory controller, L3 cache, CoreNet, and events from the other devices. The actions triggered can be actions within the other blocks of the debug unit [63].

Aurora: The P4080 implements the High-Speed IO Xilinx Aurora protocol on two lanes [75]. This allows for a maximum speed of 5 Gbit/s transmit and receive rate.

5.3. Safety-Net Processor

The safety-net processor is implemented in a Xilinx Virtex 7 XC7VX485T FPGA [100]. This FPGA features 485760 programmable logic cells, around 4 MB of memory, and transceivers capable of transmission speeds up to 12.5 Gbit/s. A simplified overview of the blocks im-

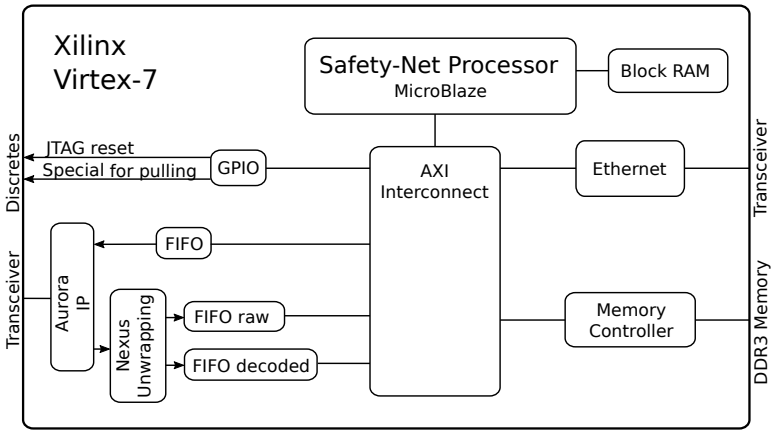


Figure 5.3.: Block diagram of the logic implemented in the FPGA.

plemented in the FPGA is shown in Figure 5.3. The overall design was created as Xilinx Vivado [104] block design while the specifically implemented IP cores are coded in VHDL. The safety-net processor is in the center of the design and is connected to the other blocks via an AXI bus. The link to the multicore processor is established via Aurora and the Nexus decoding block. In addition to the high-speed link, discrete lines are connected to the debug unit of the P4080. The received data is accessible via AXI FIFOs. Block RAM is connected to the MicroBlaze and the 1 GB DDR3 memory on the evaluation board is accessible via a memory controller. Ethernet is implemented for communication with the development PC.

5.3.1. MicroBlaze

The MicroBlaze is a 32-bit RISC soft-core microprocessor created by Xilinx [103]. It can be optimized for different parameters such as speed and size on the FPGA to fit into different designs. In this thesis, the MicroBlaze is used as the safety-net processor as a replacement for the DAL-A hardware processor. It is configured as a 5 stage pipeline running at 200 MHz with a floating-point unit and without caches. The core has access to 2 MB block RAM where the code is located. Additionally, it has access to the 1 GB of-chip memory which is used for storing the Fingerprint model. The MicroBlaze can be programmed via a debug module or the default configuration of the block ram memory in the FPGA bitstream. The interface between the MicroBlaze and other devices implemented in the FPGA is the AXI bus. Apart from the FPGA blocks explained in the following, the processor is connected to discrete lines which are connected to the P4080. These lines are JTAG and special pins of the multicore for signaling external events. The JTAG protocol is not implemented, and the JTAG pins are only used for resetting the device under observation.

5.3.2. Aurora

In order to receive the Aurora encoded data from the P4080, a Xilinx IP core Aurora 8B/10B v11.1 [101] is implemented in the design. This IP core is available from Xilinx and can be used without modification. In the design, the Aurora block is configured to use 8B/10B coding with two lanes at a total speed of 5 Gbit/s. The IP core handles the configuration of the high-speed transceiver of the FPGA. Once the devices are synchronized, the lanes are up, and data can be sent and received. The interface of the Aurora core to the other devices within the FPGA is an AXI-Stream interface. In order to send data from the MicroBlaze to the Aurora interface, a FIFO is used which handles the translation from AXI-Lite to AXI-Stream. Thus, the MicroBlaze can write to the FIFO like to a normal device by using addresses. The FIFO buffers the data and converts it to AXI-Stream which does not use addresses. Data received by the Aurora block is directly forwarded to the Nexus unwrapping IP core.

5.3.3. Nexus Unwrapping

In the NXP P series debug unit the Nexus protocol operates with a beat size of 32 bits. In every beat 30 bits are reserved for Message Data Output (MDO) and 2 bits for Message Start/End (MSE). The MDO contains the actual data to be transmitted, such as Nexus message type, timestamp and payload data (e.g. performance counter values). The MSE indicates start, end and intermediate blocks of one Nexus frame. A complete state machine describing the transitions from start to end MSE codes is given in [40].

The Nexus unwrapping IP core was created in the course of this thesis with AXI-Stream interfaces for input and output in order to provide a preprocessing of the incoming Nexus stream. The Nexus state machine was implemented in VHDL to identify the individual Nexus frames. A further decoding was implemented for message types which require a real-time processing. Messages which do not require a fast processing, such as messages used during initial configuration, are not decoded in this block and forwarded to the *FIFO raw*. The decoding of these messages is then handled in software.

The messages that are decoded are the performance counter messages and the ownership trace messages which arrive at very high frequencies. This way a high load can be removed from the safety-net processor. For these messages the Nexus frame is removed, and the performance counter values are concatenated with the most recent thread id. Therefore, every PMC value is related to a thread. The extracted PMC values with thread mapping are forwarded to *FIFO decoded* which is accessible by the MicroBlaze. It was observed that the P4080 does not send out single small messages, such as the OTM, instantly, but sends out messages in bundles. However, the order of the messages is correct and thus, the relation between the PMC and thread id is valid.

5.3.4. Ethernet

To create the model, a high amount of performance counters has to be recorded. This cannot be stored inside the FPGA as the memory is limited. Thus, a high-speed data connection to a PC has to be established to directly record the data on a hard drive. In order to provide such a connection, a Xilinx IP core AXI 1G/2.5G Ethernet Subsystem [102] was integrated in the design. This IP core handles the Ethernet framing and includes the media access control (MAC). It configures the transceiver which is connected to the Ethernet port of the FPGA development board. The IP core is configured to operate at 1 Gbit/s. The UDP/IP stack is implemented as software in the MicroBlaze. In addition to the PMC transfer during model creation, the Ethernet block is used for debugging of the safety-net system. The distinguishing between the different purposes is done by the receiving program based on the port number.

5.4. Extraction of Performance Counter Values

For the safety-net approach it is essential to extract the performance counter values from the multicore processor and send it to the safety-net processor. For this purpose, the debug unit described in Section 5.2 has to be accessed and configured in a way that a periodic high-speed low-latency access to the PMC values is possible.

5.4.1. Debug Interface Configuration Link

In order to configure the debug interface of the P4080 by an external device, JTAG and the bidirectional Nexus interface can be used. In this thesis, the JTAG pins were only utilized to reset the processor. The configuration of the debug interface is done via the Nexus back-channel. The advantage of using the Nexus interface over JTAG is that the processor does not have to be stopped for writing to memory mapped registers. The registers of the debug interface are accessible via the Nexus interface without stopping the processor. The ownership trace messages can also be configured via this interface. A Nexus software library for encoding and decoding the messages which are not covered by the created Nexus IP core was created in the course of this thesis.

Unfortunately, the performance counter registers inside the cores are not accessible via memory mapped registers in the P-series processors [72]. Only the individual cores can access the registers for configuring the specific events to be tracked by the counters. Access via memory mapped register is however possible in the T-series [73]. In order to overcome this problem in the P4080, the *instruction jamming unit* provided by the debug interface is used. With this module, instructions can be injected in a specific core and the PMC configuration registers can be written. However, this is only possible in the case of a stopped core and thus intrusive to the application running on the core. Still, this is much less intrusive than using JTAG as only the configuration of the core registers is intrusive. The initial configuration of

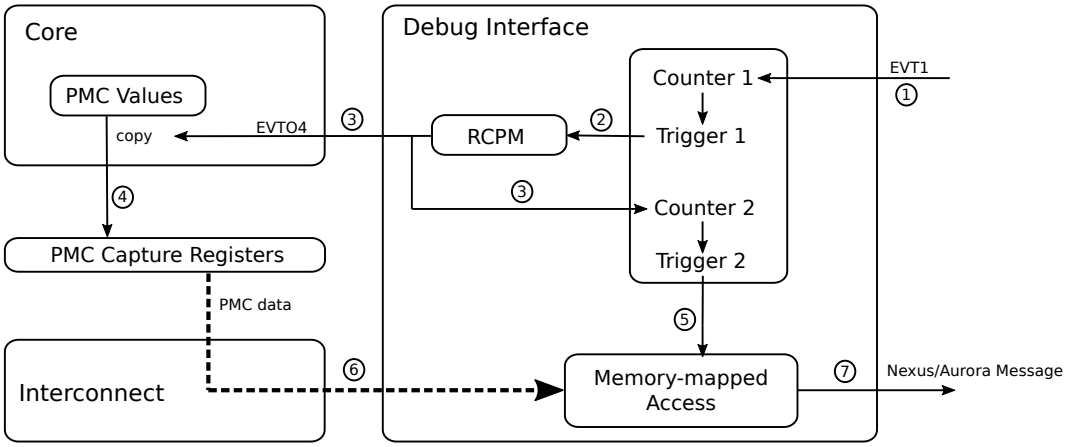


Figure 5.4.: Extraction process of the performance counters in the cores and possible interference channels (dashed line) on the example of one core of the NXP P4080.

the performance counter events and the configuration of Bus Interface Unit (BIU) watchpoint messages for the interference core detection is performed with the instruction jamming. This has to be done only once during the startup phase of the whole system and enables the configuration of the needed registers without modifying the operating system.

5.4.2. Extraction Process

The performance counter registers are not directly accessible by a memory mapped access. Therefore, an extraction process was implemented. After the initial configuration, the process of extracting the performance counter values from the cores is triggered by the FPGA. Thus, the readout frequency can be adjusted dynamically by the safety-net processor. Figure 5.4 shows the extraction process. The trigger from the FPGA is on the discrete line EVT1 attached to the debug port, at location (1) in the figure. This trigger is routed to counter 1 which is configured to directly trigger the RCPM (2). Afterwards, the RCPM triggers the event EVTO4 via a watchdog event in the relevant core (3). At the same time, this event is fed back to counter 2. The event EVTO4 is configured to copy the current PMC values to the PMC Capture Registers (4). In contrast to the PMC configuration registers, the Performance Counter Capture Registers are memory mapped and thus accessible via the debug interface. In order to read the data, a memory mapped access is triggered in step (5). This trigger results from step (3) but is delayed by counter 2 in order to ensure that the copy of the PMC values is finished before the access to the registers (6). In order to increase the readout efficiency, two of the PMC registers are accessed simultaneously. For the four performance counter values two Nexus messages are created holding two PMC values each and are sent to the FPGA via the Aurora interface (7). The procedure except for step (6) is assumed to be non-intrusive to the applications running on the multicore processor because the debug events are sent via dedicated lines. The memory mapped access, however, is intrusive as it uses the

common interconnect to read the values. An analysis on the intrusiveness is performed in Section 6.4.4.

The ownership trace messages are sent in parallel to the PMC readout process. Every time the Process ID register in the core is changed, a Nexus message is generated and sent to the Aurora port. A similar procedure applies for the extraction of BIU values of the low priority cores. Whenever a defined overflow of the BIU performance counter occurs, a Nexus message is sent.

5.4.3. Maximum Achievable Read-Out Speed

In order to determine the maximum achievable read-out speed in terms of performance counter values per second, the overhead of both, Aurora, the link layer protocol, and the Nexus [40] protocol which is transferred over the Aurora link, have to be analyzed. Aurora is implemented with two lanes on the P4080, which leads to a total theoretical bandwidth of 5 Gbit/s. However, due to protocol overhead and 8B/10B coding, the effective Aurora bandwidth is around 3.5 Gbit/s [75]. Furthermore, the Nexus protocol is implemented inefficiently (see [97]). Small chunks of data (256 bit) are sent as a message instead of long data packets. The Nexus protocol overhead is 6 times 32 bit values with meta data for transmitting 64 bit of actual performance counter values. This results in around 875 Mbit/s maximum actual data throughput. According to Equation 4.2, 128 Mbit/s of throughput is necessary for observing one core, with four PMC registers of 32 bit at a sample rate of 1 μs. Therefore, even though the protocol overhead is very high, the resulting throughput is sufficient for observing multiple cores in parallel, even at very high readout speeds. The limiting factor is not the data transfer but the capabilities of processing this data in real-time in the safety-net processor.

5.5. Model Creation

The complete process of the model creation is shown in Figure 5.5. In the first step, the application that shall be observed is executed with different input parameters and the resulting performance counter readings are transmitted to the development PC via Ethernet as shown in Figure 5.3. For this purpose, the safety-net processor only executes the initialization of the debug unit and afterwards forwards the PMC values received in the *FIFO decoded* to the Ethernet port. The data is recorded by the PC as a binary file which contains a complete stream of performance counters over the recorded time regardless of the different threads executed in that period. In the next step, a simple program was developed to transfer the raw data to a format that is readable by Octave. Octave is an open-source replacement of MATLAB where most parts of the language are compatible [22]. It has the advantage over programming in c that many algorithms are already implemented and easily accessible. For example the plotting of curves is a very simple task. This is desirable during the development

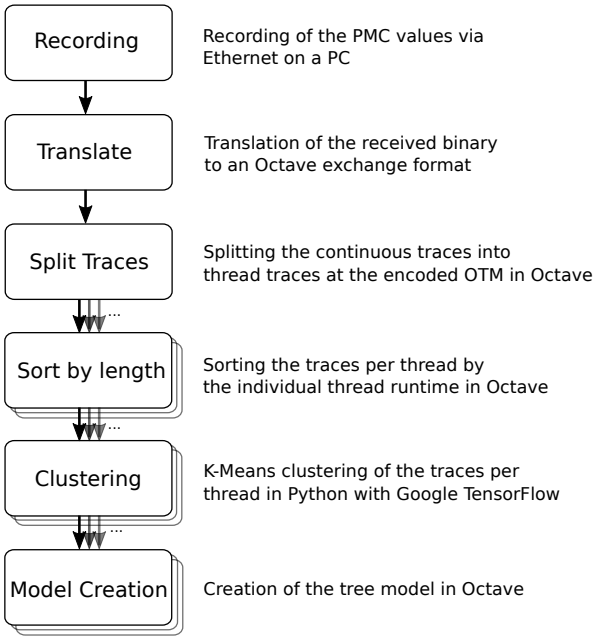


Figure 5.5.: Model creation process including the used tools. After the trace is split into different streams for the different threads (Split Traces at OTM) the following steps are individually performed for every thread.

of the approach. For a production environment, however, an implementation in C/C++ is interesting as for bigger models the processing in Octave has disadvantages with regards to performance.

Once the data is readable by Octave, an algorithm splits the traces according to the tasks based on the ownership id encoded in the trace. The further processing after this step is performed separately for each task. For every task, the corresponding curves are sorted by length to prepare for the clustering process as described in Section 4.5.2. The clustering is programmed in Python due to the availability of clustering libraries. The output format of Octave can be imported in Python. Once the clustering is done and the data is reduced to the medians of the clusters, the creation of the model is again implemented in Octave. The result of the process is a Fingerprint model per task. Every model is encoded as Octave exchange file for simulation and as .c and .h ready to be inserted into the microcontroller code.

5.5.1. Clustering

For the clustering of the curves, the bisecting k-means algorithm was implemented as explained in Section 4.5.2. For this purpose, the Google TensorFlow library [1] in Python was used. TensorFlow is an open source software library for high performance numerical computation initially designed for machine learning. It is able to use hardware accelerators such as GPUs for a fast processing. Another advantage of this library is the ecosystem with many

open source algorithms already implemented.

The center of the k-means algorithm is the distance function that determines the assignments of curves to clusters. As described in Section 4.5.2 the sum of squared errors is the default distance function. However, for the Fingerprinting this produces inaccurate results. For the approach in this thesis, the distance function in Equation 4.4 was implemented in TensorFlow. In this library, in the first step a graph is prepared which defines the operations to be executed.

```
1 a = tf.sub(curves, centroids)
2 b = tf.abs(a)
3 c = tf.greater(b, limit)
4 d = tf.cast(c, tf.float32)
5 distances = tf.reduce_sum(d,2)
```

Listing 5.1: TensorFlow graph for the bisecting k-means distance function.

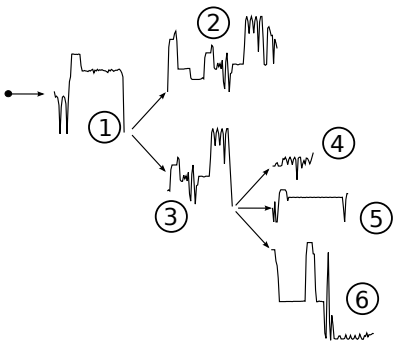
In a later step the graph is actually computed. The graph for the implemented distance function is shown in Listing 5.1. The parameters for this graph are the curves and the centroids. The curves are represented by an $n \times m$ matrix with the number of different curves n and data points within the curves m. The centroids are a $2 \times m$ matrix with the different data points m in the centroids. Since it is a bisecting k-means algorithm, where in every step the curves are assigned to exactly two centroids, n is equal to 2.

In the first step (line 1), all the curves are subtracted from both centroids and the absolute values are taken (2). The result is a $2 \times n \times m$ matrix. Afterwards, the differences are evaluated with a limit which is defining the hull around the centroid (3). The limit is set individually per performance counter based on the maximum height of a PMC recording. The result is a matrix occupied with true or false depending on the result of the evaluation. After casting the Boolean values to Float (4), which leads to ones and zeros in the data, the third dimension is reduced as individual data points are summed up per curve (5). This shrinks the $2 \times n \times m$ matrix to $2 \times n$ matrix with a distance information for every curve to both centroids.

5.5.2. Model Creation and Encoding

After the clustering, the median curves are combined into a model according to Algorithm 3 described in Section 4.5.3. One node consists of at least 4 samples. The models are encoded in three arrays: flow, catalog and data.

Flow: The flow represents the tree as adjacency matrix (shown in Figure 5.6). Every row represents a node. The edges are represented by the columns of the node. In the matrix, the edges are augmented with probabilities (Markov chains). For example, the first row/node has edges to node 2 and 3 as both columns are greater than zero. The probability of node 2 is 0.3 while the probability of node 3 following after node 1 is 0.7. The probability information




```
flow = {{ 0,.3,.7, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0},
        { 0, 0, 0,.1,.4,.5},
        { 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0}}
```

Figure 5.6.: Example of a Fingerprint model encoded as adjacency matrix with edge probabilities.

is extracted from the clustering depending on the number of curves in the respective cluster. An adjacency matrix has several advantages. First of all, it is very simple to implement and is very similar in Octave and C code. Hence, it can be very easily exported to the safety-net processor. Additionally, access to the data is possible in $\mathcal{O}(1)$. The drawback of adjacency matrices is the required storage, but since only the flow is stored as adjacency matrix and not the data, it can be neglected. Even though adjacency matrices provide the possibility of describing graphs, the implementation in this thesis is a tree due to the simpler model creation process.

Catalog: In addition to the flow, a catalog array is implemented which defines the possible nodes. Every node is described in the catalog as a section (index of begin and end) in the data array. Therefore, identically shaped curves can be reused in different nodes to save storage.

Data: In the data array the actual data points of the curves of the model used in the comparison are stored. These curves are then only stored once in the data array but referenced multiple times in the catalog array.

5.6. Interference Detection

The interference detection is implemented as C code on the MicroBlaze soft core processor. The software is executed bare metal without any operating system. A simplified flowchart is shown in Figure 5.7. At the start of the safety-net system, the debug unit of the P4080 is initialized and configured as described in Section 5.4.1. Afterwards, the interference detection is executed in a loop with a period equal to the sampling period. All the computation for the detection and triggering of the counter measures has to be completed within this loop. This restricts the number of parallel observable paths in the model. The first action of every loop iteration is the reading of the new PMC values from the FIFO implemented in the FPGA. If no values are available in the FIFO, an error is raised and a fault on the multicore processor is assumed. In the next step, the reading of new PMC values is triggered via GPIOs of

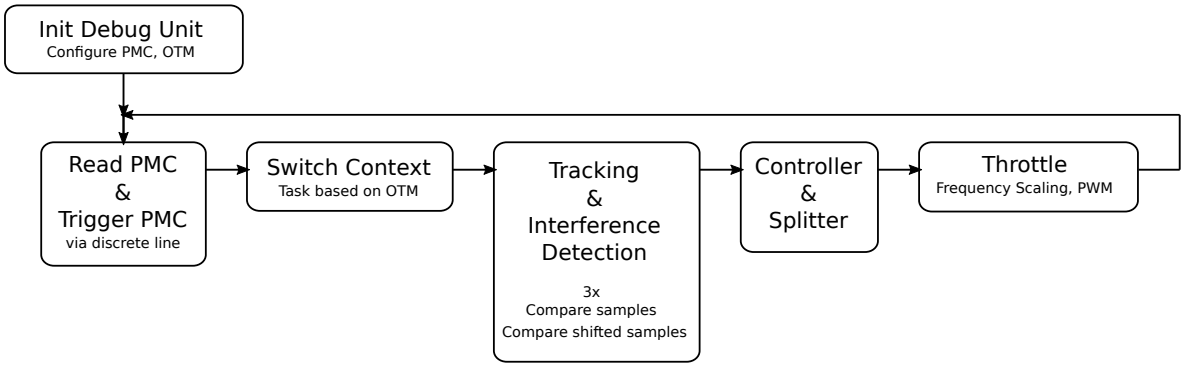


Figure 5.7.: Flowchart of the software executed on safety-net processor. The loop after the initialization is running in a 100 μs period.

the FPGA to the discrete EVT1 line of the P4080 (see Figure 5.4). This starts the readout process on the multicore processor and the new values are available in the next loop iteration. The readout process is not triggered directly before the readout of the FIFO because the new values are available with a latency, and stalling the MicroBlaze for these values would be a waste of processing time.

The next action in the loop is the context switch between the observation of different threads/partitions. In the safety-net processor, context data such as the current delay, the current node in the model, etc. are stored for every observed thread. These contexts are switched based on the ownership trace message encoded in the PMC values (see Section 5.3.3).

The tracking and interference detection is the most time-consuming block in the loop. In order to keep the loop cycles of 100 μs, a maximum number of three parallel paths is tracked simultaneously. For every path the Euclidean distance function (Equation 4.6) is computed for a δ of four samples. Hence, the most recent four measurement points are compared to four values in the model. This is done individually for each of the four performance counter events. The comparison is not only done for individual nodes but also for concatenations of nodes in case of an edge in the model. It is possible that the four model values are concatenated of three samples of one node concatenated with one sample of a successor node. Additionally, the dissimilarity is determined for a displacement d of 1 sample. The measurement values are also compared to the model curve which is shifted by one sample. Hence, a total of 24 comparisons is computed (two per PMC event and path with four PMC events and three model paths). In case the displacement yields better comparison results over three subsequent measurement points, the delay is added to the global delay and the displacement of 1 is defined as the new reference.

5.7. Controller

The controller is calculating the amount of throttling to be applied to the cores and uses the output of the interference detection (see flowchart in Figure 5.7). In the first step the threshold based on the progress d_p is calculated (see Equation 4.9 in Section 4.8). Since t_{scWCET} and d_a are constant values per thread, the threshold is calculated as

$$d_p = C1 \cdot t_r + C2 \tag{5.1}$$

with constants $C1 = \frac{d_a}{2(t_{scWCET}+d_a)}$ and $C2 = \frac{1}{2}d_a$. These constants are pre-calculated and stored in the context of the corresponding thread. Afterwards, the error e of the threshold d_p and the output of the interference detection, the measured absolute delay d_m, is calculated according to Equation 4.10. In the final step, u_{total} is determined from the error (Equation 4.11) as

$$u_{total} = 1 - C3 \cdot e \tag{5.2}$$

with constant $C3 = \frac{m}{d_a}$. This result is the output of the controller and the input of the splitter which determines the throttling per individual core. The amount of throttling per core is then implemented for every core as

$$u_{core} = (2 \cdot u_{total} - 1 + i_{core}) \cdot I_{DAL,core} \tag{5.3}$$

with the interference core detection level per core i_{core} and the DAL of the application (converted to a numeric representation according to Section 4.8). Since the equations contain many constants, the calculation of the throttling executes very fast on the safety-net processor.

5.8. Throttling of Interfering Cores

The final software block in Figure 5.7 is the throttling of the cores. In this block, the throttling techniques *frequency scaling* and *PWM*, as described in Section 4.7, are implemented. The update of the new frequency or the halt/continue signal happens once in a loop time.

The main device for throttling within the debug unit of the multicore processor is the *Run Control and Power Management (RCPM)*. In order to write to the registers of this device, the Nexus messages are encoded in software on the safety-net processor. This introduces a slight overhead in the processing compared to a hardware wrapper implementation of the Nexus protocol but is much easier to implement. The resulting messages are shown in Figure 5.8 on the right entering the RCPM together with the clock distribution scheme. The chosen Nexus message does not need an acknowledgment from the receiving device.

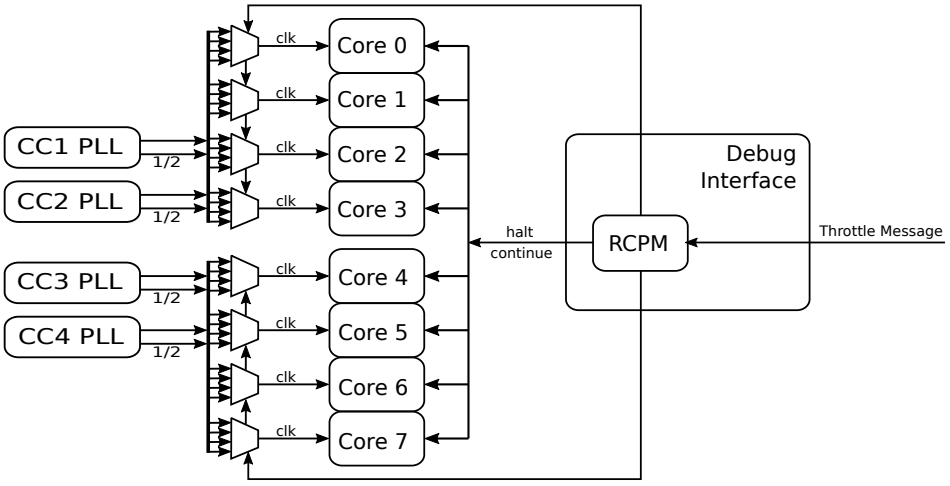


Figure 5.8.: Clock selection infrastructure of the NXP P4080 [74]. The selection of the clocks can be done by the RCPM during runtime, individually per core.

5.8.1. Frequency Scaling

The internal clocking architecture of the P4080 is shown in Figure 5.8. There are four different PLLs: CC1 to CC4. The resulting frequency of each PLL is routed to a multiplexer in two ways: the actual frequency and the frequency divided by 2. The configuration in the RCPM defines the routing of the clocks to the individual cores. However, CC1 and CC2 can only be routed to the cores 0 to 3 while CC3 and CC4 can be routed to the cores 4 to 7. In order to have a maximum frequency range, the highest selectable clock frequency is 1.5 GHz (set to CC1 and CC3) while the lowest frequency is 800 MHz (set to CC2 and CC4). Therefore, the possible frequencies are 400 MHz, 750 MHz, 800 MHz and 1.5 GHz which can be configured for each core individually. In the safety-net throttling software block, the selection of the frequency is done linearly according to the corresponding u_{core} and updated in every loop cycle. A message is only sent in case the values for the particular core has changed.

5.8.2. PWM Halt - Continue

Aside from the frequency scaling, the RCPM can also halt and continue individual cores as shown in Figure 5.8. For this purpose, it is possible to set individual cores to the *debug halted* mode. It is also possible to continue the execution via the RCPM. A PWM period of 1 ms was chosen for the implementation which is equal to 10 times the 100 μs required to track the application performance. Hence, it is possible to reduce the performance of cores competing with the main core in steps of 10 %. During the duty cycle, all the opponent cores are active, while for the rest of the period the cores are stopped (see Section 4.7). Similar to the frequency scaling, the PWM duty cycle is calculated linearly from the u_{core}. In case of a task switch, the current progress in the PWM cycle is stored for the old task, and the

progress in PWM cycle of the new task is loaded from the context. This allows to throttle tasks with switches faster than the PWM cycle of 1 ms accurately.

6. Evaluation

Different aspects of the safety-net approach as well as the full control loop are evaluated based on the implementation presented in Chapter 5 in order to show the applicability to different types of applications, the performance, and the influence of the design parameters. Due to hardware restrictions, different aspects are evaluated in different environments. The different parameters of the interference quantification, throttling, and the closed loop controller are analyzed and their contributions to the five main objectives of the safety-net, as presented in Chapter 4, are discussed.

This chapter is structured as following: In Section 6.1 the software under observation as well as the software running on the opponent cores in the course of this evaluation are presented. The different environments in which the software is executed are presented in Section 6.2. Afterwards, the evaluation results for the hybrid environment, the full system integration including a non-intrusiveness analysis and a real-world application environment are presented in sections 6.3, 6.4 and 6.5. Finally, in Section 6.6 the results are discussed.

6.1. Software Executed on the Multicore Processor

In order to evaluate the safety-net approach, an application has to be executed on the multicore processor which shall be observed. Four applications were created for two different purposes. The first purpose is to run an avionic application or an application that is similar to a real avionic application in terms of runtime, periodicity, etc. A real-world helicopter application was used for this case. However, this application can only be executed in a specific environment (see Section 6.2.3). Thus, an application based on the TACLeBench benchmark suite was created to simulate an avionic application which can be executed in any environment. The second purpose is the creation of a worst-case scenario application which acts as an opponent to show the feasibility of the approach even in the presence of very high interferences.

6.1.1. TACLeBench

TACLeBench [23] is a benchmark suite comprising five packages of algorithms which are commonly used in embedded systems. In this thesis, the TACLeBench (version 1.9) sequential package is used because these algorithms do not fit completely in the private caches like

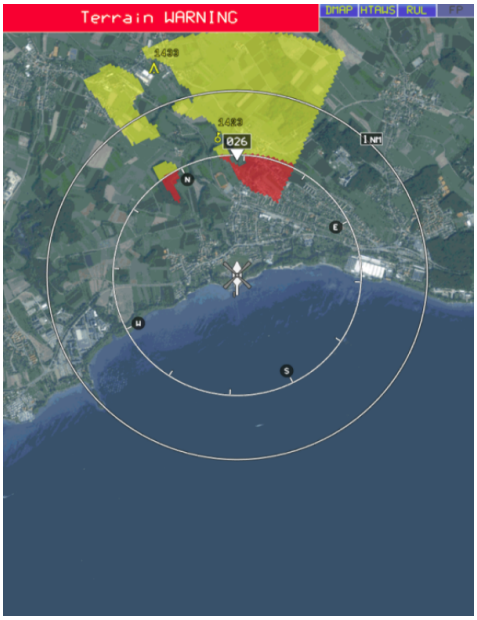


Figure 6.1.: Pilot interface of a helicopter application. Terrain that is higher than the current altitude is highlighted. The current position of the helicopter is displayed in the center of the picture. [52]

the other benchmark packages. From the sequential package, 19 of the 23 algorithms are selected. The residing four are not used because these rely on a math library which is not available for the bare-metal execution on the P4080. Examples of the selected algorithms are encrypting, sorting, H.264 block decoding and image recognition. These programs can be compiled independently of standard libraries and operating systems, which makes them easy to adapt to the test system. The code size of the individual algorithms ranges from 117 to 2710 lines of code. The complete list of the selected algorithms is given in Table A.1.

For evaluation purposes, the TACLeBench algorithms are used in two different test applications. In the first application, the 19 benchmark algorithms are executed successively in a loop according to the order given in Table A.1. In the following, this application is referred to as *static TACLeBench*. In the second application, the order of the algorithms is defined randomly at the start of every loop cycle. Thus, a program that acts differently for different input parameters is created to simulate an avionics application that behaves differently for different sensor values. This is named *dynamic TACLeBench* in the rest of the thesis.

6.1.2. Real-World Helicopter Application

For the evaluation of the approach, a real avionics application developed by Airbus was used. This application is a pilot support system showing the current location of the helicopter on a map including additional information such as high buildings, power transmission lines and way points of a predefined flight plan (see Figure 6.1). Furthermore, areas on the map with a

Partition	Duration (ms)	Max. obs. num. memory accesses
A	4	2764
B	4	7381
C	16	477886
D	10	262962
E	4	4275
F	16	477886
G	4	7020
H	8	6618

Table 6.1.: List of partitions scheduled on the helicopter application including duration and maximum number of observed memory accesses as published by Agrawal et al. [4].

higher altitude than the current altitude of the helicopter are displayed in red and yellow to prevent the pilot from crashing into mountains. This application supports the pilot in flying during night time, in bad weather conditions and poor visibility, rough terrain and at low altitudes. The information displayed is based on the current position, heading and altitude of the helicopter as well as a database with maps and obstacles.

It is a DAL-C application and was originally developed for a single-core processor. This legacy application was ported to run on one core of a multicore processor. It consists of scheduled tasks within time and space partitions. The software is implemented using the real time operating system *Wind River VxWorks 653* [99] as IMA system with multiple partitions. The operating system was extended so that the current task id is written to the *Nexus Process Id register (NPIDR)*. Thus, the *Ownership Trace Message(OTM)* needed for distinguishing the partitions and threads, explained in Section 4.4.4, can be used. The IMA schedule has a fixed major cycle of 66 ms and is divided in the partitions listed in Table 6.1. As visible in the table, the partitions C, D, and F execute the highest number of memory accesses and are therefore prone to interferences and slowdowns by other cores. The input of the application is the position, heading and speed of the helicopter. Depending on these values, different paths in the program are executed. For example the resulting output screen is different for flying over water or in alpine terrain.

6.1.3. Read/Write Opponent

In order to create a close to worst-case interference scenario benchmark, as much data as possible has to be stored/loaded to and from memory. In the presence of caches, every access to the memory should be a cache miss in the private caches in order to create interferences. Therefore, the distance between two consecutive memory accesses has to be at least the size of a cache line. In the case of the P4080 this is 64 bytes in the L1 and L2. This technique is described in more detail in [71]. For the evaluation, an assembler program was developed

that consists of a loop that only executes either load or store instructions with a distance of 64 bytes. Thus, the code fits into the instruction cache but not in the data cache.

6.2. Environments

The described applications are executed in three different environments. This is due to the fact that not all of the aspects of the safety-net can be fully evaluated in one environment; the implemented MicroBlaze safety-net processor, for example, has performance restrictions. Thus, evaluations that require a higher amount of performance are done in simulation. Another reason is that the helicopter application was ported to a specific dual-core processor. The three environments presented in the following are hybrid environment, full system integration, and the real-world application.

Furthermore, as described in Section 5.2, the P4080 implements different levels of cache. Depending on the configuration of these caches the amount of interference is different. For the evaluation different configurations of the caches (see Table 6.2) were used to demonstrate the different technologies under appropriate conditions. In all configurations the private L2 cache is disabled because enabling it increases core-local caches and reduces interferences between cores (due to lower miss-rates). However, a higher amount of interferences is desirable to show the full capabilities of the safety-net system. Moreover, L3 is never used as shared cache since this would complicate a possible WCET analysis. The different cache configurations used during the evaluation are shown in Table 6.2. The *Realistic* configuration uses the local L1 instruction and data caches and the external SDRAM as main memory. *Max. Interference* also disables both L1 caches to generate the maximum accesses from the cores to the interconnect. Since all accesses target the external SDRAM, they show a comparatively long latency. The *Max. Traffic* configuration is similar to *Max. Interference* but uses an internal SRAM (L3 used as SRAM) instead of the external SDRAM. This configuration generates the highest traffic on the interconnect due to the low latencies of accesses.

Realistic		Max. Interferences		Max. Traffic	
L1	on	L1	off	L1	off
L2	off	L2	off	L2	off
L3	off	L3	off	L3	int. SRAM
Memory	ext. SDRAM	Memory	ext. SDRAM	Memory	int. SRAM

Table 6.2.: Different cache configurations used in the evaluations.

6.2.1. Hybrid Environment

The hybrid environment consists of an NXP P4080 multicore processor attached to a Xilinx Virtex-7 FPGA as described in Chapter 5. However, in this environment the safety-net

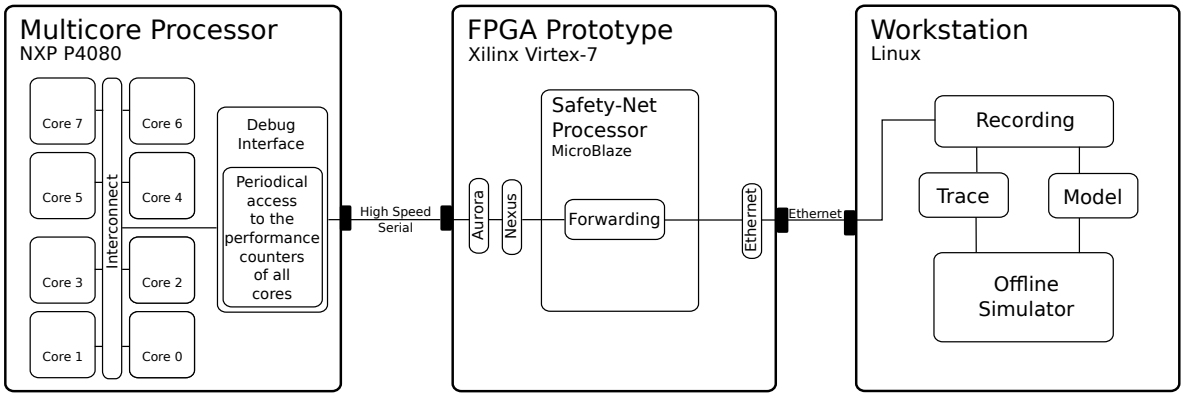


Figure 6.2.: Hybrid environment with the data generation on the P4080, extracted by the Virtex-7 and forwarded to a workstation. The data processing is done offline on the workstation.

processor is used for initialization of the multicore processors debug unit and for forwarding the received performance counter values. For this purpose, an Ethernet stack is implemented on the MicroBlaze. The interference detection algorithms as well as the controller are not executed on the MicroBlaze. Instead, interference detection is executed in an offline simulator on a Linux workstation (see Figure 6.2). The executed code in the simulator for the tracking of the application according to the model is identical to the code on the safety-net processor. Since the data traffic received on the Ethernet port of the workstation is low enough to be continuously written to the hard drive (see Section 5.4.3), a high number of performance counter values can be recorded. At the very high sampling rate of 1 μs the recorded data is 15.26 MB/s for four PMCs. In one hour, this results in around 55 GB.

The hybrid environment is needed for the evaluation of cases where a very fast read-out speed is needed because the MicroBlaze can only perform a continuous interference detection on data arriving with a sample period greater than 100 μs. However, the main drawback is that there is no mitigation of slowdowns possible. The controller and the throttling mechanisms cannot be simulated because in this case, the behavior of the P4080 had to be modeled.

6.2.2. Full System Integration

The full system integration environment is shown in Figure 6.3 and represents the implementation described in detail in Chapter 5. In this environment, the NXP P4080 is observed by the Virtex-7 FPGA. The interference detection, controller and throttling are executed in real-time on the safety-net processor and only the model creation is done offline. On the P4080, the TACLeBench suite as well as the read/write opponents can be executed for evaluation purposes. An Ethernet connection from the safety-net processor to a workstation is used for transmitting the evaluation results.

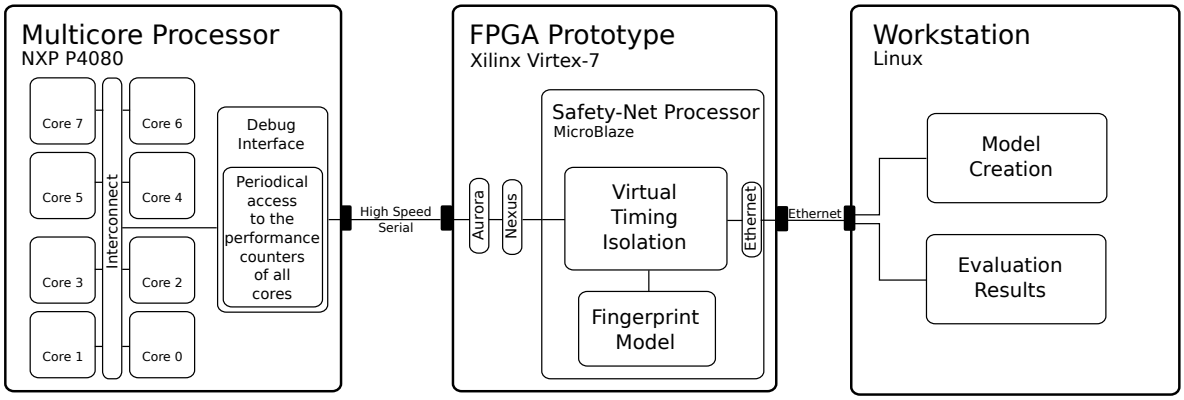


Figure 6.3.: Block diagram of the safety-net hardware setup used for the P4080 development system in the full system integration environment.

6.2.3. Real-World Application

The hardware environment of the real-world application mainly consists of four parts: The multicore processor, a graphics processing unit, data storage for the map data, and an FPGA (see Figure 6.4). The processor executing the application software is the dual core processor NXP P5020. This processor implements two e5500 64 bit PowerPC cores. The devices on the SoC, the performance counters and the debug unit are very similar to the P4080. Only slight adaptions to the configuration algorithm in the safety-net processor had to be done in order to interface this processor. The helicopter application is executed on one core (core 0) while the other core (opponent core) can be equipped with another application (e.g. read/write opponent, TACLeBench). During the evaluation, a workstation executing a simulator constantly provides the input parameters position, heading and altitude of the helicopter. This replaces the real helicopter avionics data. The application configuration (e.g. selection of map layers) is also done on the workstation. Furthermore, the application has access to a hard drive with map data to identify the obstacles. The GPU is used for the graphical output.

In the Xilinx Kintex-7, which is a midrange FPGA, the safety-net processor was implemented as explained in Chapter 5. The implementation is similar to the P4080 implementation. The FPGA has less logic cells than the Virtex-7 but is still sufficient. The high-speed serial interface of the FPGA is connected to the Aurora/Nexus interface of the P5020. The safety-net processor is linked to the workstation via UART because in this case Ethernet was not available. The interface is used for recording the trace data on the workstation and creating the model. Unfortunately, the throttling of processor cores is not possible because no reliable Aurora back-channel is available due to the hardware design. Configuration of the debug registers can be done but no real-time throttling. Therefore, it is possible to record PMC data, create a model and detect interferences, but it is not possible to close the loop.

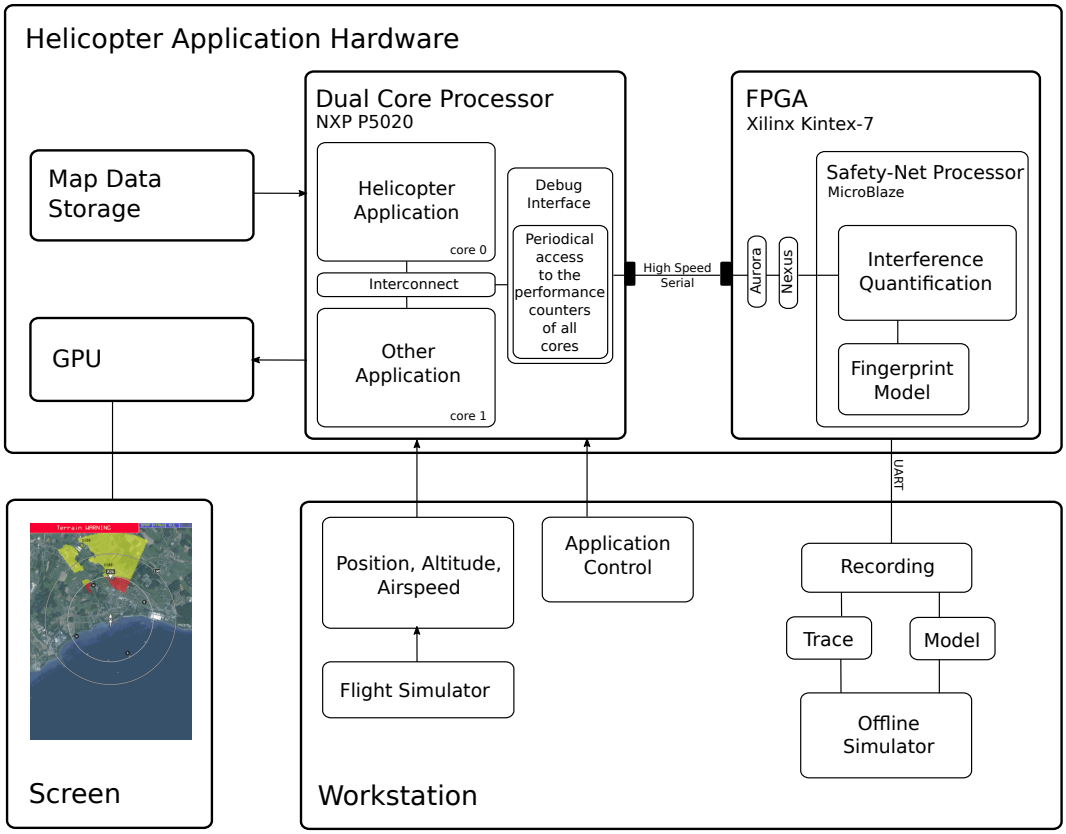


Figure 6.4.: Block diagram of the safety-net hardware setup used for the helicopter application IMA system [52].

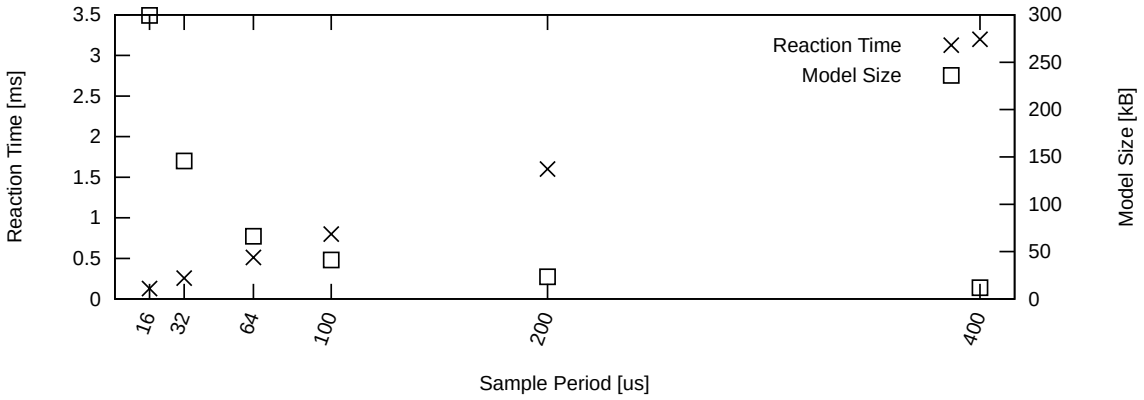


Figure 6.5.: Reaction time and model size of the interference quantification algorithm depending on the sample period. The experiment is based on the *dynamic TACLeBench* execution with the *realistic* cache configuration and four *read* opponent threads on the other cores. The tracked performance counter event is *Instructions completed*.

6.3. Evaluation in the Hybrid Environment

In the hybrid environment, two evaluations were performed: Different sample rates including the implications on the detection speed, and the effect of different performance counter events on the quantification precision.

6.3.1. Sample Rate

For the sample rate experiment, the *dynamic TACLeBench* was executed on the P4080 and sampled by the safety-net in six different periods from 16 μs to 400 μs . The multicore was set to the *realistic* cache configuration (see Table 6.2).

For all the different sample rates, the following steps are performed individually. First of all, 10 seconds of consecutive TACLeBench executions (around 40 ms each, i.e. 250 iterations) are recorded in standalone mode and a model is created. Afterwards, four opponent threads are executed on the other cores to create an average slowdown of 5 % on the TACLeBench executions which are again recorded. With the model and the slowed down recordings, the interference quantification is done in the simulator which is identical to the interference quantification code executed on the MicroBlaze. However, in the simulator, the quantification does not have to be performed in real-time and higher sample rates can be processed. The results for the different TACLeBench executions within the 10 seconds are averaged.

The quantification precision, or how well the interfered curves matched to the model at the individual sample periods, was similar for all the executions. However, the time after which a slowdown could be detected, and thus the minimal reaction time, was highly dependent on the sample rate. The results for the different sample rates are shown in Figure 6.5. The reaction

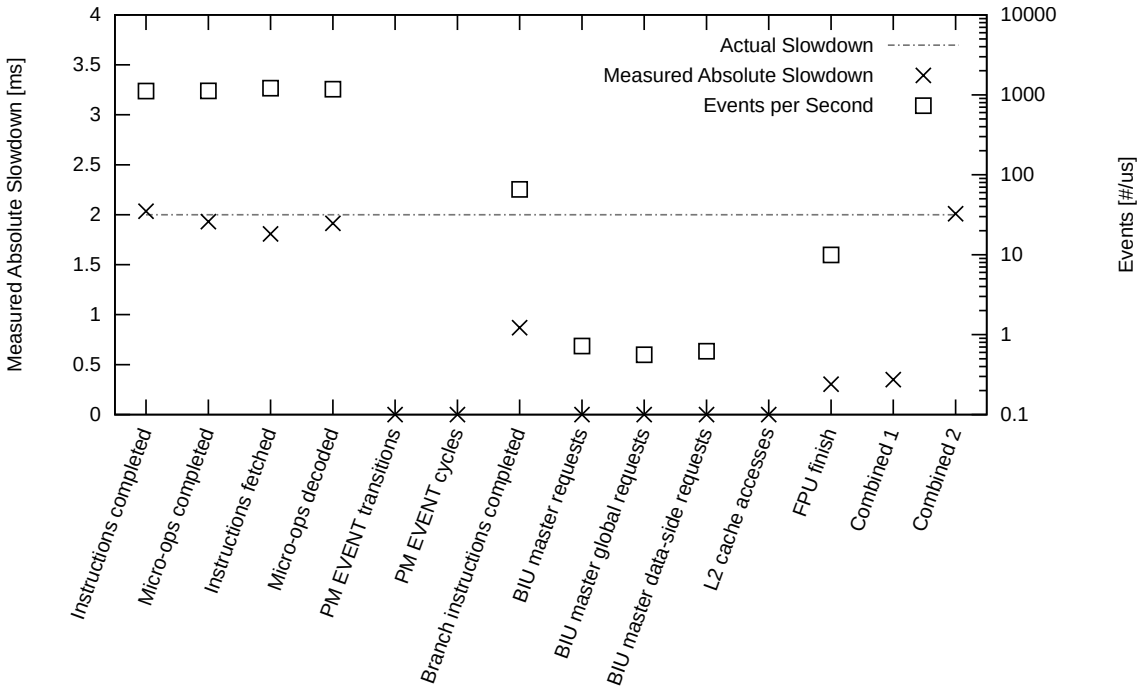


Figure 6.6.: Interference quantification accuracy depending on the performance counter events used for the tracking. The experiment is based on the *static TACLeBench* execution with the *realistic* cache configuration, four *read* opponent threads on the other cores, and a sample period of 100 μs.

time is given in milliseconds and scales linearly from 0.13 ms in case of 16 μs to 3.21 ms in case of 400 μs sample period. The model sizes are growing naturally linearly with the sample frequency. Thus, it decreases exponentially with the sample period. However, since the model not only has to include more sampling points due to the lower period but also more program paths are stored in the model which are averaged out for lower sampling periods, the model size increases additionally for smaller periods. With the increasing number of samples to compare and the number of paths in the model, the processing power needed to compute the quantification scales similar to the model size. Therefore, the sample period should be chosen depending on the reaction time constraints of the application which depends on the size and period of the time partitions.

6.3.2. Performance Counter Events

In order to evaluate which performance counter event types are suitable for interference quantification, 86 of the 180 different performance counter events selectable in the P4080 were analyzed. The other 94 events were not analyzed in detail because these do not represent a program run, e.g. event 1: *Processor cycles* or event 82: *PMC0 overflow*. For the analysis,

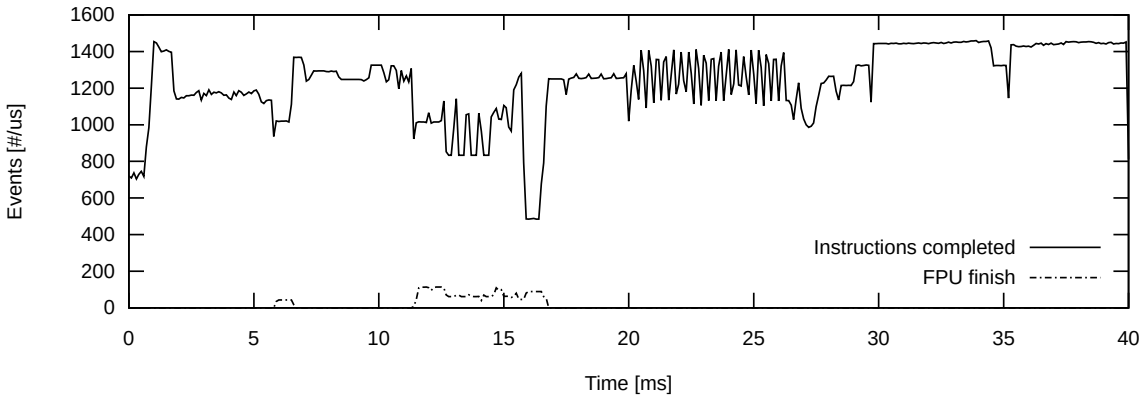


Figure 6.7.: Comparison of the *Instructions completed* and *FPU finish* performance counter events of one run of the *static TACLeBench* at a sample period of 100 μs.

the *static TACLeBench* was executed for 10 seconds on the P4080 with the *realistic* cache configuration in standalone and recorded with a sample period of 100 μs. From this data, the model was computed for every PMC event individually. Afterwards, the benchmark was executed with four *read* opponent threads which lead to an absolute slowdown of 2.0 ms (42.2 ms total execution time per benchmark run instead of 40.2 ms). In simulation, the measured total slowdown of the application was determined by tracking the recorded slowed down curves with the model.

A selection of the 86 analyzed events is shown in Figure 6.6 including the measured absolute slowdown. The ideal measurement is 2.0 ms as this is the real total slowdown. A measurement higher or lower indicates an insufficient tracking and a low quantification precision. As visible in the figure, tracking with the event *Instructions completed* leads to very good results whereas tracking with the event *FPU finish* alone leads to a very low quantification precision.

An explanation for this behavior is given in Figure 6.7. There, the curves for the events *Instructions completed* and *FPU finish* during the execution of a TACLeBench are displayed. The event type *Instructions completed* constantly produces a high number of events per second whereas the *FPU finish* event type only raises a low number of events during the execution of specific algorithms. Therefore, for *FPU finish*, the tracking algorithm has no data available which could be used for the comparison to the model in order to determine the interference.

This relationship is visible in Figure 6.6 where the events per second are displayed in addition to the measured slowdown. For a high number of events per second (e.g. around 1000 events per microsecond) the quantification precision is very high. For low numbers of events per time unit, the curves are not usable for the tracking.

In addition to the analysis of individual event types, combinations of four event types were evaluated with the same parameters as for the individual executions. The following two examples are shown in the figure:

- Combined 1 (A combination of events with a low number of events per second): *FPU finish*, *BIU master requests*, *Load micro-ops completed*, *Data L1 cache locks*
- Combined 2 (A combination of a mixture of events with a low and high number of events per second): *Instructions completed*, *Branch instructions completed*, *BIU master requests*, *FPU finish*

The analysis of *Combined 1* yields that the combination of these low frequent events does not significantly increase the quantification precision when compared to the individual events. For *Combined 2*, the analysis shows that the quantification precision could be slightly increased compared to the *Instructions completed* individual execution. The main contribution is brought by the *Instructions completed* PMC, but the low frequency events increase the robustness of the tracking. For example, a branch with an equally high number of *Instructions completed* can be distinguished by the *FPU finish* if the branches differ in that manner.

6.4. Evaluation in the Full System Integration Environment

In the full system integration environment, the complete loop can be evaluated. Furthermore, the accuracy of the slowdown quantification, which acts as the sensor input to the controller, and the effectiveness of the throttling, which acts as the actuator, are evaluated. The complete loop is evaluated with different controller algorithms. This section concludes with a non-intrusiveness analysis to show that the safety-net itself does not create noticeable interferences. In contrast to the real-world application, an advantage of the full system integration environment is that the software running on the P4080 can be modified according to the needs for the evaluation.

6.4.1. Interference Quantification Accuracy

In order to rely on the output of the interference detection, based on which of the low priority cores are throttled, the accuracy of the interference detection has to be evaluated. This is done by comparing the actual slowdown of an application measured on the P4080 to the slowdown determined by the safety-net interference quantification.

For this purpose, the TACLeBench suite was instrumented. The instrumentation is inserted at the start and after every benchmark. Thus, 20 milestones are inserted into the 19 algorithms of the *static TACLeBench* suite. Each instrumentation consists of a time measurement within the multicore processor, which is stored in the RAM of the P4080 for later readout, and a trace message, which is sent to the quantification FPGA to be mapped to the interference quantification. Therefore, the time measurements can be used to calculate the actual slowdown which can be compared to the interference quantification values at the time these messages are received.

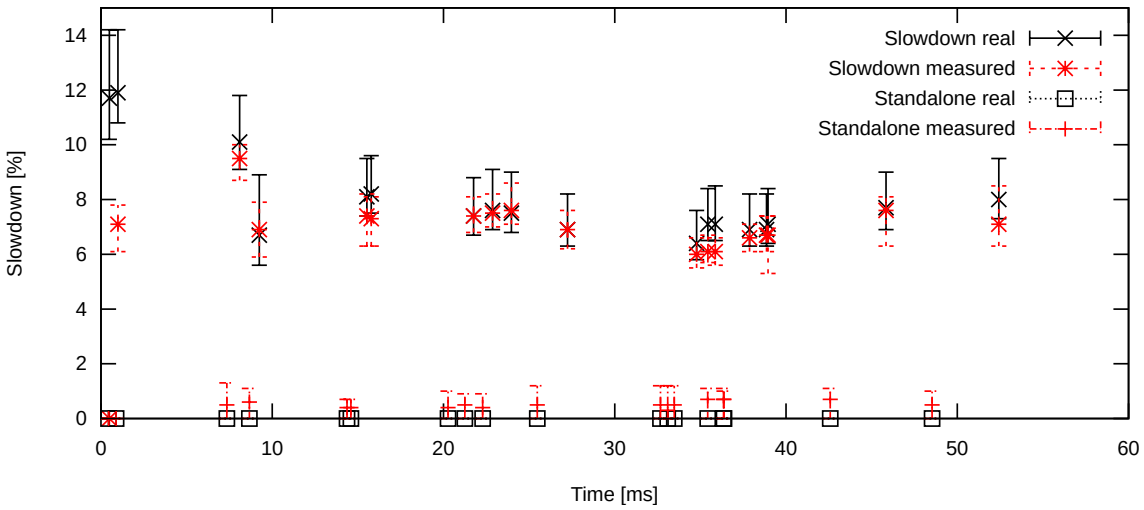


Figure 6.8.: Quality of the quantification over the runtime of the static TACLeBench suite in standalone and with seven opponent cores (*Slowdown*) using the *Realistic* cache configuration (L1 cache is enabled). The plotted dots represent the mean values while the bars reflect the minimum and maximum value measured. The standalone executions end at 48.5 ms while the slowed down executions end at 52.4 ms, which is a total slowdown of around 8 %.

For the comparison, the TACLeBench was first executed standalone in order to record the time measurements without slowdown as shown in Figure 6.8 "Standalone real". All the different measurements are performed 100 times and the mean values are plotted. The bars indicate the minimum and maximum values within the 100 measurements. The cache configuration in this experiment is *Realistic* (L1 cache is enabled). At the same time, the slowdown was measured by the FPGA displayed in the "Standalone measured" curve. Here it is visible that there is a slight overestimation in the interference quantification of around 1 % in some cases.

In the second step, the benchmark was executed with seven *write* opponents causing an average total slowdown of around 8 %. However, the slowdown over time varies as it is depending on the different algorithms and their memory access behavior. The actual slowdown is shown in Figure 6.8 "Slowdown real" while the slowdown detected by the interference quantification with Fingerprints is labeled "Slowdown measured". Overall, the real and the measured slowdown are matching very well. For example, the final (at 52 ms) actual average slowdown value is 8.0 % compared to a measured slowdown of 7.1 %. In the case of the "Standalone" execution, the final average overestimation of the slowdown is only 0.5 %. In total, the average deviation of the average value is less than 1 %. However, in the beginning of the run (first millisecond) the values do not match. This is due to the fact that the quantification algorithm takes a small start period of around 1 ms to align the measured

curve with the curves in the model. However, once this alignment is fixed, the matching is very responsive as can be seen in the figure.

6.4.2. Throttling Effectiveness

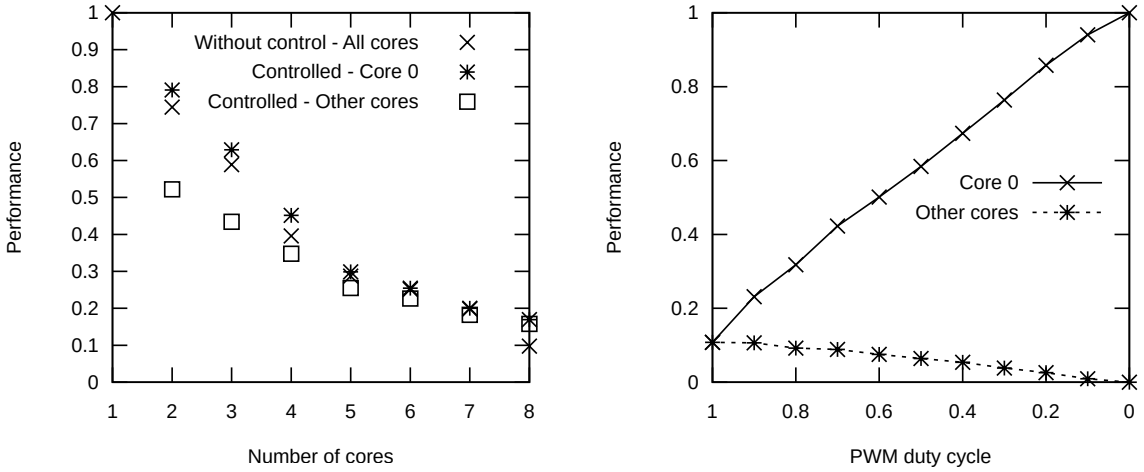
In case the control values result in a throttling of a low priority core, it has to be ensured that the throttling options are effective. Thus, both the PWM and the frequency scaling approach have to be evaluated. The *static TACLeBench* and the read/write algorithm was used for this purpose in three different scenarios:

1. Read with seven read opponents (Figure 6.9): shows the worst-case interference scenario,
2. TACLeBench with seven write opponents (Figure 6.10): shows a realistic application (this benchmark is application oriented and generates realistic traffic on the shared interconnect and memory and profits from local data caches) with worst-case opponents,
3. TACLeBench with seven TACLeBench opponents (Figure 6.11): shows a realistic application on core 0 with realistic opponents.

All the scenarios were evaluated in two different cache configurations: *Realistic* (L1 is enabled) and *Maximum Interferences* (no caches enabled). For the evaluation of the frequency scaling, the performance of the applications was measured without frequency scaling of the opponent cores in the first step. All cores were running with 1.5 GHz which is labeled *Without control* in the figures. In scenarios one and three the performance is identical for all applications on all cores as the applications are identical. In scenario two, the *Without control* performance is depicted individually. In a second step the opponent cores are set to 400 MHz, which is the minimum configurable speed for the P4080, when the maximum speed is configured to 1.5 GHz. The performance of core 0 as well as the performance of the other cores was measured. As expected, it is visible that the performance of the other cores drops while the performance of core 0 is increased. However, the amount highly depends on the scenario and cache configuration. The measurements were taken for a varying number of opponent cores. For example, in case of *Number of cores* is four, core 0 runs with 1.5 GHz, cores 1 to 3 run with 400 MHz and cores 5 to 8 are idle in the controlled case. In the case of only one core, there is no data plotted for *Core 0* and *Other cores* since there are no other cores running, and the performance of core 0 does not degrade. This data point was used to normalize the measurements.

During the evaluation of the PWM approach all eight cores are utilized while the duty cycle of all seven opponent cores varies from 0 to 100 % in steps of 10 %. As result, the execution time for the main application as well as for the opponent applications is measured.

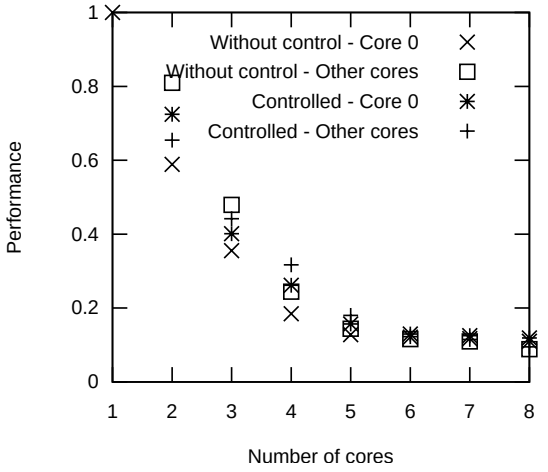
The result for scenario one with the read algorithm on the main core and up to seven read opponents is shown in Figure 6.9. It can be observed that the frequency scaling is not able



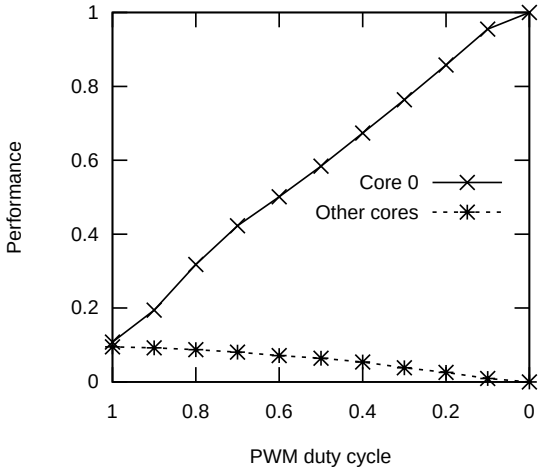
(a) Frequency scaling for a varying number of opponents. The opponents are scaled down to 400 MHz while core 0 is running at 1.5 GHz in the controlled case. Since the same software is executed on all cores, the performance is identical for all cores in the uncontrolled case.

(b) PWM halt and resume of the opponent cores for a varying duty cycle while seven opponent cores are running. For example, at duty cycle 0.2 the other core are running 20 % of the execution time while core 0 is running in 100 % of the execution time.

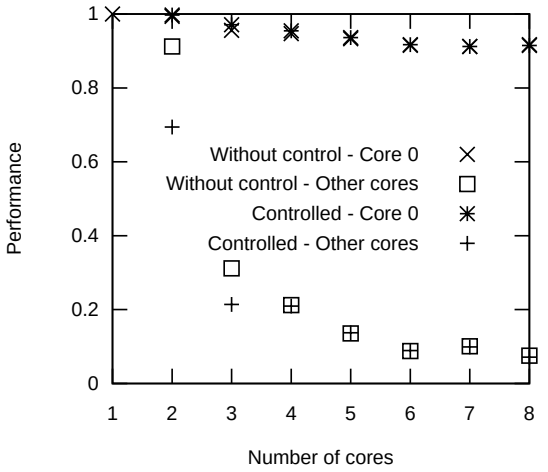
Figure 6.9.: Frequency Scaling and PWM efficiency with the read algorithm (see Chapter 6.1.3) on core 0 as well as read opponents on the other seven cores. These measurements were conducted with the *Maximum interference* cache configuration (all caches disabled). A separate analysis (not shown here) indicated that the curves are very similar for enabled local caches as the read algorithm is designed to cause maximum stress on the interconnect and does not take advantage of caches.



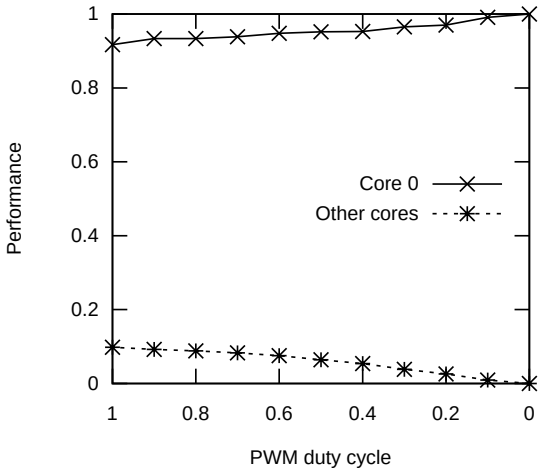
(a) Frequency scaling of the opponents to 400 MHz. No caches enabled.



(b) PWM halt and resume of seven opponents. No caches enabled.

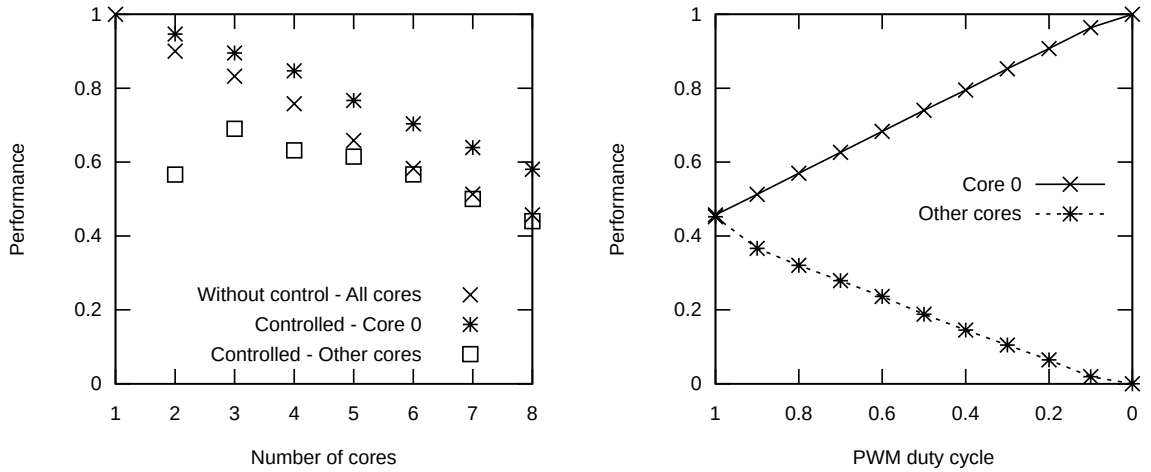


(c) Frequency scaling of the opponents to 400 MHz. L1 cache enabled.



(d) PWM halt and resume of seven opponents. L1 cache enabled.

Figure 6.10.: Frequency scaling and PWM efficiency with TACLeBench on core 0 and write opponents on the other cores. For Figure **a** and **b** the *Maximum interference* cache configuration applies while for Figure **c** and **d** the *Realistic* cache configuration (L1 cache enabled) was used.



(a) Frequency scaling of the opponents to 400 MHz.

(b) PWM halt and resume of seven opponents for a varying duty cycle.

Figure 6.11.: Frequency scaling and PWM efficiency with TACLe on core 0 and TACLe opponents on the other cores. The measurements were taken with the *Maximum interference* cache configuration.

to reduce the interferences from the opponent core enough to keep core 0 at a performance level higher than 90 %. The increase of performance in comparison to the uncontrolled case is only around 4 % for one opponent core (number of cores equal to two) and is completely negligible for seven opponent cores (number of cores equal to eight). This effect is the same for both cache configurations and can be explained by the cache behavior of the algorithm. Even though the opponent cores are executing instructions with one fourth of the speed, the memory interface is still jammed by the opponents because the memory is even slower. In contrast to that, the results for the PWM approach shown in Figure 6.9b reveal that even in the case of running seven opponents in parallel, the performance of the main application can be fully recovered. This shows that in the worst case the frequency scaling is not sufficient, but the PWM approach can control the performance of the main application at the cost of heavily slowing down the opponents.

The more realistic scenario of TACLeBench with seven write opponents is shown in Figure 6.10. In contrast to the other scenarios, there are four curves displayed for the frequency scaling instead of three. This is because there are different applications running on core 0 and on the other cores which are evaluated separately. In the case of no caches, the results are similar to the results in the *read/read* scenario. However, if the L1 cache is enabled, the performance of the TACLeBench does not drop below 90 % even with seven *write* opponents. The effect of the frequency scaling is not significant because of the cache behavior of the *write* algorithm like in the *read/read* scenario.

For the scenario of TACLeBench with seven TACLeBench opponents, the results are dis-

played in Figure 6.11. It is visible that the frequency scaling has a significant effect on the performance of the application on core 0. Especially in the case of one and two opponents (two and three cores in Figure 6.11a), the frequency scaling increases the performance to over 90 %. However, even though a performance increase of around 15 % compared to the uncontrolled case is visible in the eight core case, frequency scaling is not sufficient for advancing the performance to a level of over 90 %. Additional measurements (not shown in the figure) show that in the case of the *Realistic* cache configuration the loss in performance of the TACLe benchmark on core 0 is negligible and the performance of core 0 with seven opponents is still 99 %.

Concluding this evaluation, frequency scaling is less efficient for improving performance of core 0 compared to PWM. On the other hand, frequency scaling affects applications running in parallel to core 0 less than PWM.

6.4.3. Controller

The controller closes the loop between the interference quantification and the throttling of the interfering cores. The efficiency of its algorithm allows for a maximum runtime of the low priority core while ensuring that the high priority core finishes before its deadline. The following three controller approaches were evaluated:

- **Threshold-based controller**
  Stops the concurrent cores when the slowdown of the main application exceeds a given threshold and enables the cores again when the slowdown falls below the same threshold again (see Figure 4.14b).
- **Proportional controller**
  Throttles the interfering cores with an actuator based on the PWM activity control and the frequency scaling. The higher the slowdown, the higher the throttling on the interfering cores.
- **Progress aware controller**
  Applies PWM throttling and frequency scaling while taking the progress and deadline of the application into account, according to the algorithm described in Section 4.8.

To analyze the effectiveness of the different controller algorithms, the *static TACLeBench* was executed as the main application and the *write* algorithm as opponents running on seven cores in parallel. The benchmarks were executed on a P4080 with the *realistic* cache configuration and recorded with a sample period of 100 μs. A model of the TACLeBench was created in standalone mode which was used for the tracking.

In standalone, the benchmark finishes after approximately 40 ms. This is assumed to be the single-core WCET (t_{scWCET}) (see Section 4.1). The deadline was set to 43 ms while the acceptable delay d_a was defined as 2 ms (5 % of t_{scWCET}) in order to allow for a 1 ms

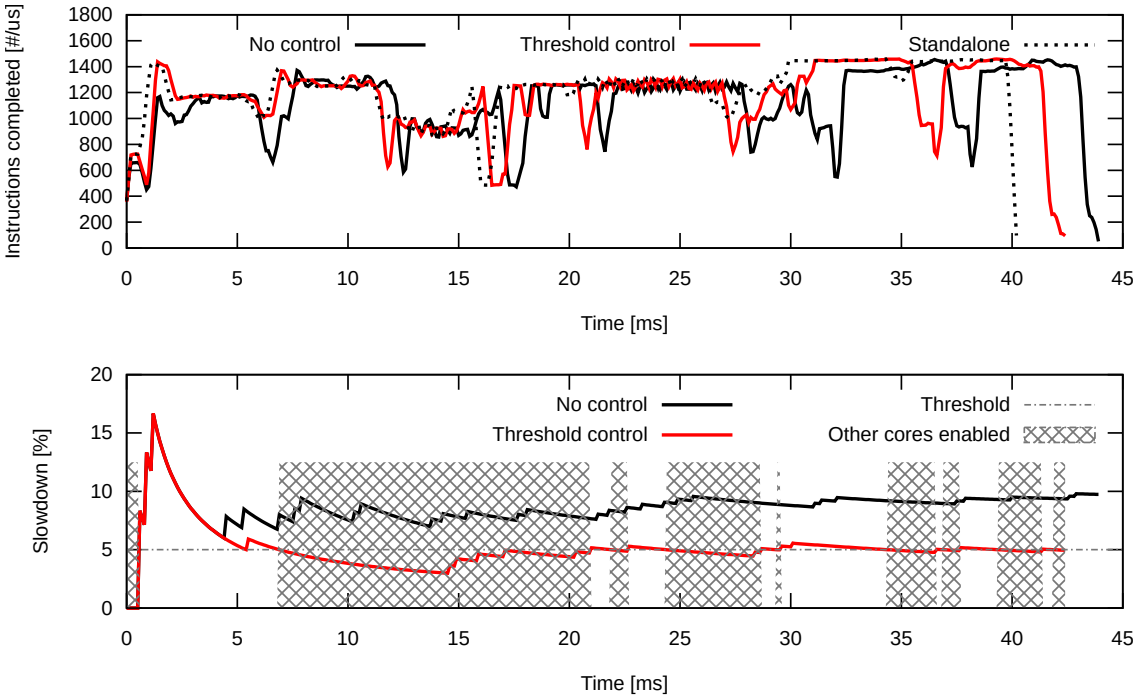


Figure 6.12.: Threshold-based controller with halt and continue. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores.

safety-margin to the actual deadline. The goal of the controller algorithm is to ensure that the benchmark finishes within 42 ms. For the progress aware controller, the constants $C1$ to $C3$, defined in Section 5.7, are therefore $C1 = \frac{d_a}{2(t_{scWCET}+d_a)} = 0.024$, $C2 = \frac{1}{2}d_a = 1ms$, and $C3 = \frac{m}{d_a} = 5ms^{-1}$ with a margin m of 10 %. Since the interference from the opponent cores is identical for all the cores because the same software is executed, the output of the controller, the throttling u_{total}, is equally distributed to the interfering cores while the critical application is untouched.

The results of the evaluation are shown in the figures 6.12 to 6.16. There, the progress of one example *static TACLeBench* over time is displayed in the upper part and the measured slowdown over time in the lower part. The upper part presents the number of executed instructions per microsecond. For comparison, the standalone (no opponent applications) and the uncontrolled (seven opponent applications without control of the safety-net system) executions are displayed. The uncontrolled execution takes about 10 % (4 ms) longer than the standalone run due to the interference. The diagrams in the lower part of the figures represent the slowdown of the main application as tracked by the *Fingerprinting*. Since the tracking of progress is based on discrete steps, the performance reductions are manifested in sharp steps. The following phases of smooth performance increases are caused by relative distribution of a slowdown over a longer time, i.e. a one-time delay at the start of the application of 5 % is

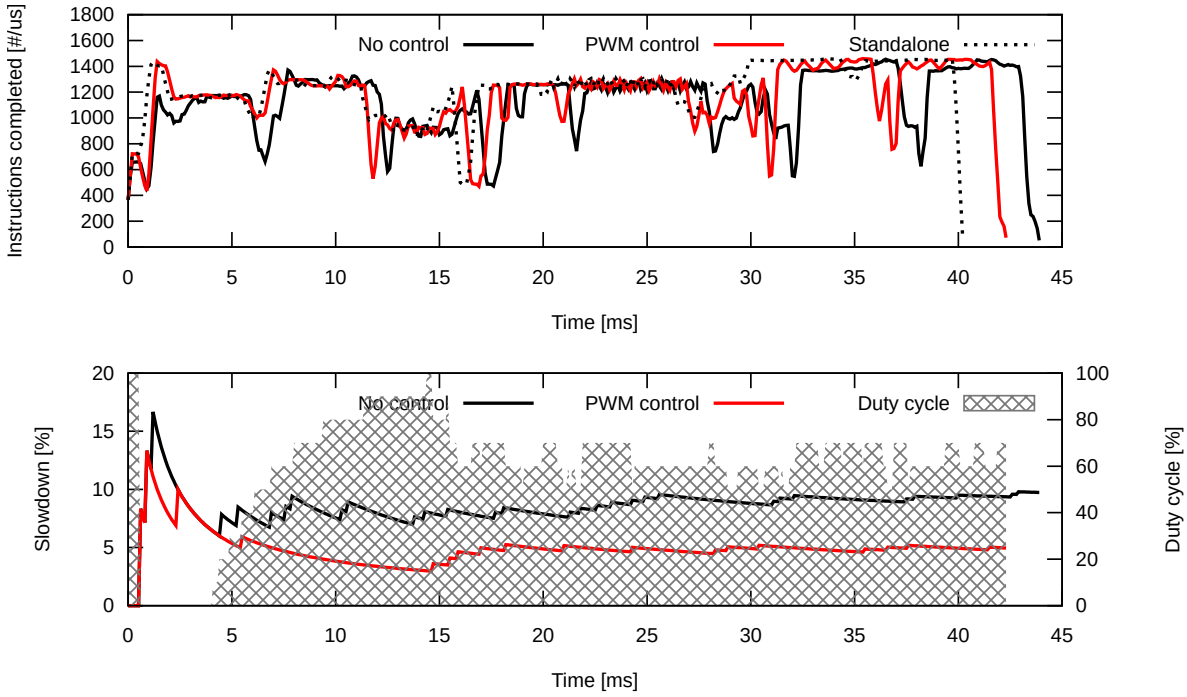


Figure 6.13.: Proportional controller with PWM. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores.

reduced over the total execution time to a much lower slowdown.

The results of the threshold controller are displayed in Figure 6.12. The dotted line in the lower diagram represents the threshold (5 %), i.e. the maximum target slowdown of the main application. The gray shaded boxes indicate the times when the other seven cores are active. No gray shading means that the other cores are disabled by the control mechanism. It is visible that the opponent cores are disabled whenever the measured slowdown is higher than the threshold value, which keeps the total slowdown in the end at a measured slowdown of 4.95 %. The actual total slowdown is 5.22 % (measured by comparing the times it took for executing the benchmark in the standalone and controlled case), which means an underestimation of the slowdown by the *Fingerprinting* and an exceedance of the threshold by less than 0.5 %. During the total run of one TACLeBench benchmark the opponents are executed for 60.2 % and halted for 39.8 % of the time.

The behavior of the PWM controller is shown in Figure 6.13. The duty cycles of the competing cores are set according to the measured slowdown. A slowdown of less than 2 % allows full performance for all cores, a slowdown above 7 % leads to completely disabled competing cores. Between 7 % and 2 %, the duty cycles are adjusted in 10 % steps from 10 % to 90 % (one step per half percent of slowdown). The gray shaded areas represent the duty cycles of the PWM core activation signal. As can be observed, the 5 % target slowdown of the main application was slightly missed after completion (4.96 % measured while the

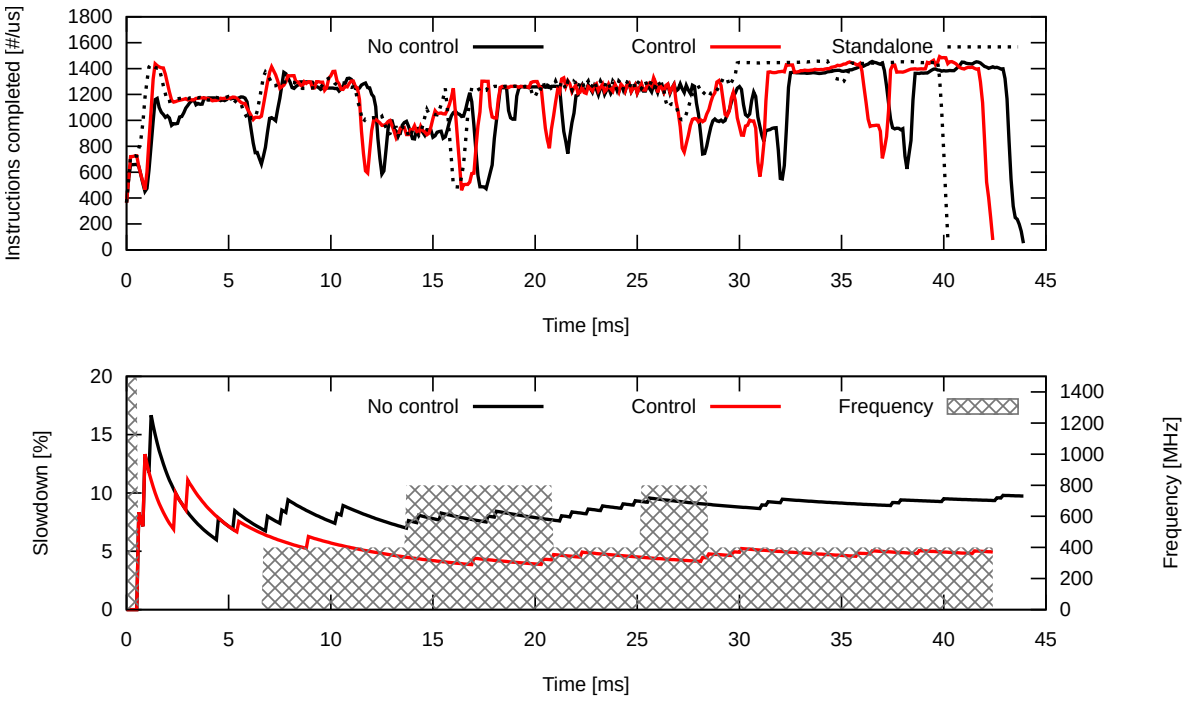


Figure 6.14.: Proportional controller with frequency scaling. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores.

actual slowdown was 5.02 %). In this case, the controller reaches the targeted measured goal (4.96 %) while the actual target slowdown was missed due to the inaccurate measurement of the interference detection. However, this inaccuracy is covered in the margin between the acceptable delay and the actual deadline (see Section 4.1).

The active phases of the competing cores are much longer in time but less intensive. A PWM signal is used, which means that the cores are active for many but smaller periods. With this PWM control, the seven *write* opponents get 61.4 % of the cores' performance while the main application still meets the performance requirements.

In the course of the evaluation, different controller limits (e.g. 3 % to 8 % instead of 2 % to 7 %) were evaluated. However, the before mentioned configuration leads to the best results in terms of reaching the target slowdown while achieving the highest possible throughput on the other cores.

The frequency scaling approach is displayed in Figure 6.14. The possible frequencies of the opponent cores are 400 MHz, 800 MHz and 1.5 GHz. Furthermore, the core can be halted. Similar to the PWM approach, a slowdown of less than 2 % allows full performance for all cores, a slowdown above 7 % leads to completely disabled competing cores. Between 7 % and 2 %, the frequencies are adjusted in linear intervals. The gray shaded areas represent the frequencies of the opponent cores. The slowdown of the main application is reduced with a

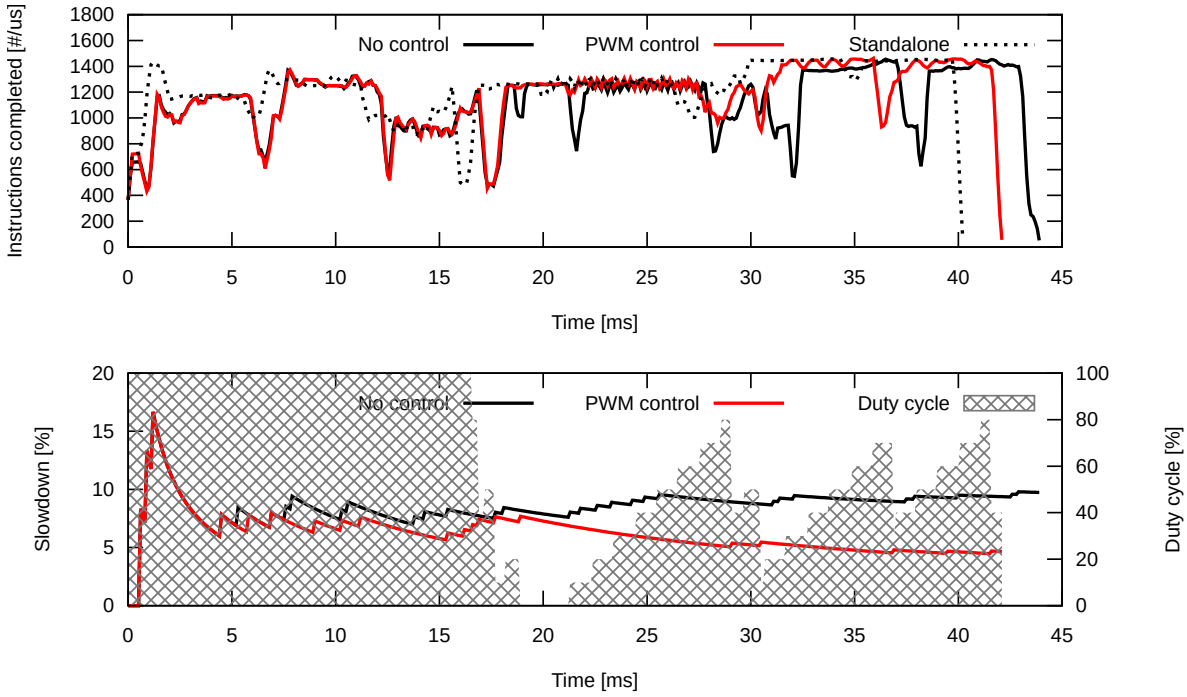


Figure 6.15.: Progress aware controller with PWM. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores.

total measured value of 4.93 % (real slowdown: 4.96 %). However, this was not possible by only scaling down the cores. During the period of high interference in the beginning of the execution the opponent cores had to be halted for a sufficient reduction of the interferences. An assessment of the cores' processing time compared to the aforementioned approaches does not make sense in this case. The frequency scaling of a core cannot be compared with halting and continuing a core because the performance of a scaled down core is highly dependent on the instructions executed. For example, if many ALU operations are executed, the program execution is much more delayed than in case of program sections with load/store operations where the processor is more likely to stall as an effect of the executed instructions. Thus, the executed amount of ALU operations per second is reduced while the amount of load/store operations per second is almost identical when the frequency is reduced.

Similar to the proportional controller, the progress aware controller algorithm was evaluated with both PWM and frequency scaling. The results are shown in figures 6.15 and 6.16. The characteristic of the progress aware controller is visible as in the beginning of the execution no throttling is performed until a certain amount of absolute slowdown is accumulated (at around 17 ms). Thus, the other cores are not directly stopped in the beginning where the application is fetching big amounts of data from the memory, as explained in Section 4.8. The closer the execution is to the acceptable delay, the more aggressively the algorithm throttles the other cores. For the case of PWM as the throttling method, the measured slowdown is 4.64 % while

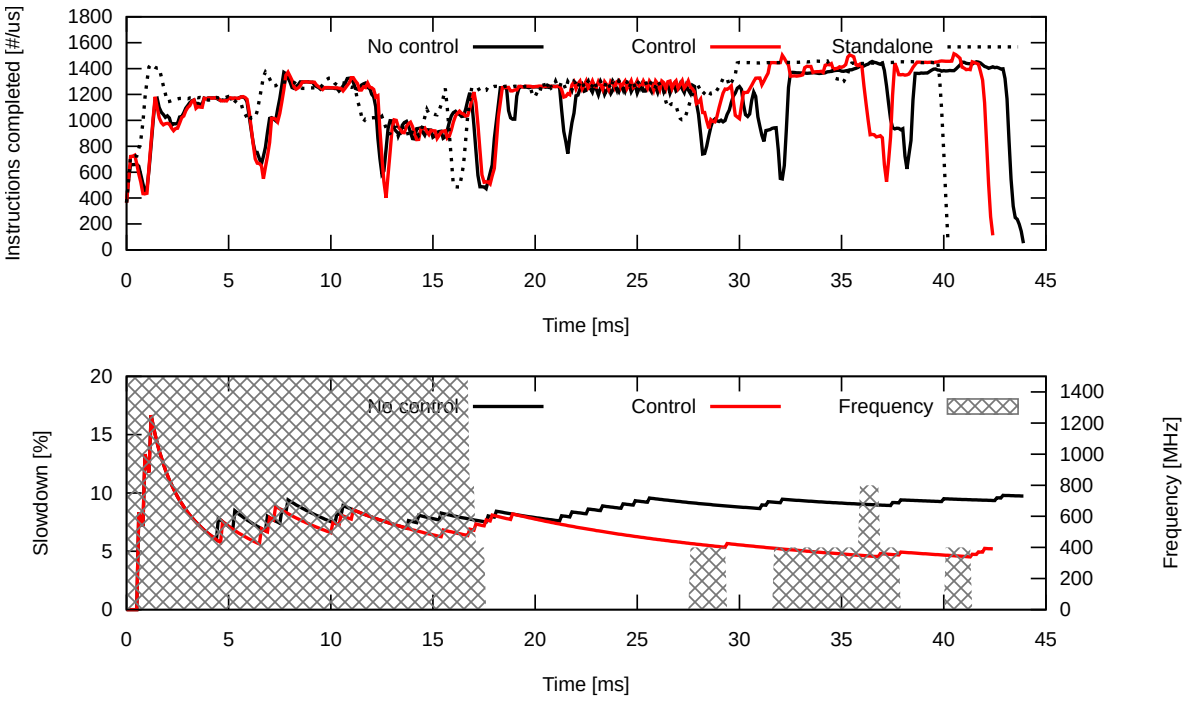


Figure 6.16.: Progress aware controller with frequency scaling. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores.

the actual slowdown is 4.71 %. The opponents get 64.1 % of the cores' performance. Thus, in comparison to the threshold and proportional controller, the progress aware controller yields better results. If frequency scaling is used for the throttling, the program slightly misses to finish within the acceptable delay. The final measured slowdown is 5.23 % while the actual slowdown is 5.41 % even though the other cores are stopped most of the time after the slowdown accumulated to half of the acceptable delay.

In all the scenarios, the benchmark finishes before the deadline. However, frequency scaling without completely stopping the cores does not lead to a sufficient throttling in case of the *write* opponents. The main advantage of the progress aware controller over the threshold and proportional controller is that it only starts the throttling if really needed and thus enables maximum throughput on the other cores while still keeping the deadline of the high priority application. Furthermore, for the proportional controller, the upper boundary was set to 7 % instead of 5 % in order to allow for a maximum execution time on the other cores. In the case of the TACLeBench, the acceptable delay of 5 % was achieved, but a slight overshoot is theoretically possible. Thus, the progress aware controller is the safer option as the other cores are completely disabled once the acceptable delay is reached.

6.4.4. Non-Intrusiveness

In order not to create any further interference on the critical application, the read-out overhead of the progress tracking should be as small as possible, in the ideal case absolutely non-intrusive. However, as the *Fingerprinting* approach relies on the performance counter values (see Section 4.3) which reside inside the cores, these values have to be accessed from outside the SoC. The extraction process including the possible interference channels are explained in section 5.4.2. Every read-out is triggered by a signal sent by the external timing isolation system. Once an external signal is received, the first steps (1 to 4 in Figure 5.4) inside the SoC are the transfer of the performance counter values to the memory-mapped *Performance Monitor Counter Capture Registers* as the performance counter registers inside the cores are not accessible by the debug interface. Measurements showed that this happens in a non-intrusive way as no delay of a program executed on the core could be observed.

In a later step (6 in Figure 5.4) the *Performance Monitor Counter Capture Registers* are accessed by a memory mapped access. Therefore, it is assumed that the interconnect is used to transport the data to the debug interface. This is a possible interference channel as the interconnect is also used by the cores when they access the memory, the shared cache or the I/O interfaces.

For a reliable tracking, four 32 bit performance counter values per core need to be extracted as mentioned in Section 4.3. Depending on the read-out frequency, the bandwidth needed on the interconnect varies. For an example extraction frequency of 1 MHz (1 μs period), a bandwidth of 128 Mbit/s per observed core is needed. However, if the performance counters of all cores shall be extracted in parallel at this frequency, the resulting bandwidth is 1 Gbit/s. Even though NXP claims that the P4080 provides 0.8 Tbit/s coherent read bandwidth [74], interference is measurable even if only one core is observed.

The interference measured for the execution of the sequential TACLeBench benchmarks on one core while the remaining cores are idle is shown in Figure 6.17. The slowdown in percent s is defined as

$$s(p) = \left(\frac{x(p)}{x_{unobserved}} - 1 \right) * 100 \tag{6.1}$$

with the access period p, the execution time without observation $x_{unobserved}$ and the execution time for a given period for reading out the PMC capture registers $x(p)$.

The bars in Figure 6.17 are the respective observed min/max values. At some points the bars are below zero. This results from the fact that the execution time is varying even without disturbance from the read-out process. These slight variations are a result of cache, interconnect and memory mechanisms.

It can be recognized that the slowdown of around 0.09 % at access frequencies of 10 Mhz is very low. For access periods larger than 20 μs the interference, is not distinguishable from the noise. The result is identical for the case that the performance values are extracted from the

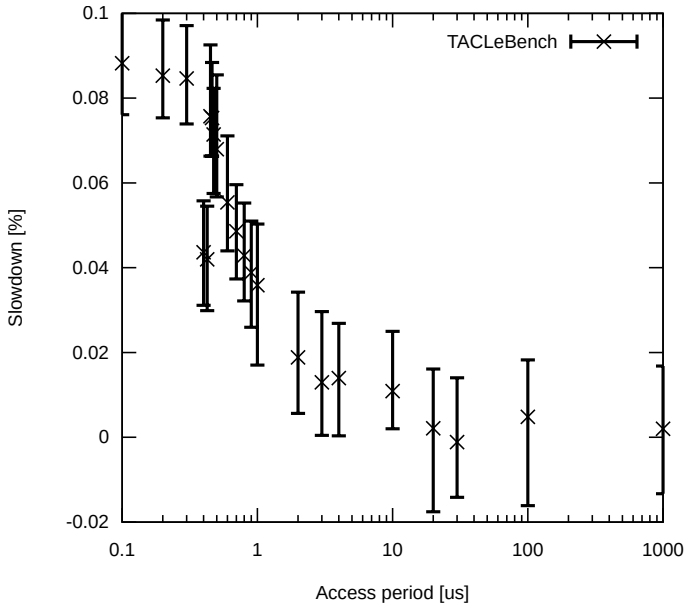


Figure 6.17.: Slowdown of the TACLeBench execution on one core depending on the access period of the performance counter read-out process.

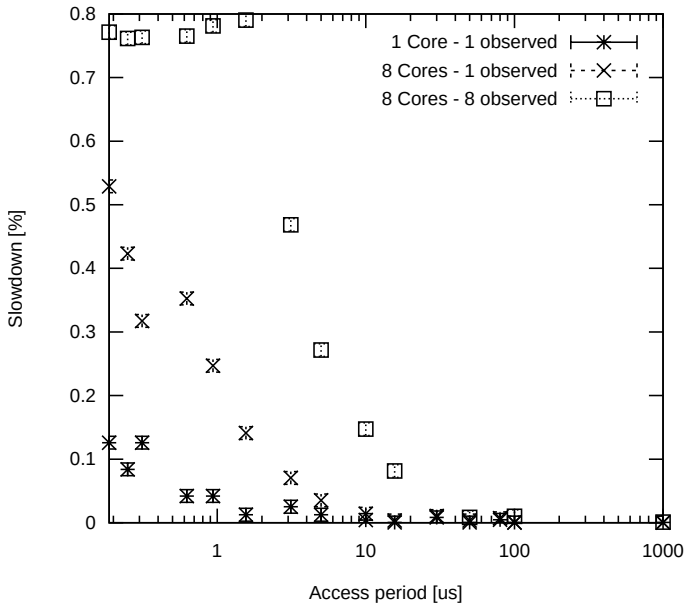


Figure 6.18.: Slowdown of the read benchmark to the on-chip SRAM (L3 cache configured as SRAM) while the L1 and L2 caches are disabled.

core executing the benchmark as well as from any core that is in idle mode. Even though the interference is decreasing with a higher access period there are two measurements that are lower than expected (around 0.4 μs). It is assumed that this is because of a synchronization of the memory accesses of the TACLeBench with the memory mapped accesses of the debug interface.

The intrusiveness analysis using the TACLeBench benchmarks shows the potential impact on a real application. However, in order to determine the worst-case interference of the read-out process, the read/write opponents were executed. Additionally, the L1 and L2 caches are disabled while the L3 cache is used as SRAM memory (*Maximum Traffic* cache configuration). Therefore, every load instruction initiates a transaction in the interconnect which is considered as the worst case.

The slowdown of this application is displayed in Figure 6.18 for three configurations. In the first configuration, the application runs on one core while the remaining cores are idle. In the second and third configuration, the application is executed on all eight cores simultaneously. The extraction process is performed on one core or all the cores. The slowdown is determined similarly to the TACLeBench analysis with Equation 6.1 but normalized to the eight core execution without reading the counter registers. Thus, the slowdown resulting from the inter-core interference is eliminated.

For the one core execution, the measured slowdown is not significantly higher compared to the TACLeBench analysis. However, when all the eight cores are used for execution, the slowdown is around six times higher if only one of the eight cores is observed, which is still a very low slowdown. The higher interference for the eight core execution results from the utilization of the interconnect from the cores. For access periods larger than 20 μs (50 kHz), the interference is again not quantifiable. In case all the cores are observed simultaneously, the interference is much higher. As expected, the interference is around eight times higher compared to the case where only one core is monitored. The slowdown reaches a maximum at around 1.02 μs, and it is not increasing with decreasing access periods. At this point, the maximum speed of the triggered memory mapped access of the debug interface is reached. However, for extraction periods above 50 μs, the interference is also not distinguishable from the noise.

6.5. Evaluation on a Real-World Helicopter Application

The helicopter application is executed in the environment described in Section 6.2.3. The input values such as position, heading, and speed are defined beforehand and are played back so that the helicopter is flying at a predefined trajectory in order to generate reproducible results. As mentioned before, due to hardware limitations, throttling is not possible. Thus, only the interference quantification is evaluated on this software. Similar to most evaluations

on the P4080, the readout speed of the PMC values is 100 μs (10 kHz).

6.5.1. IMA Applicability

Figure 6.19 shows the performance counters of one major IMA cycle with indicated time slots splitting in slots A to H (mentioned in Table 6.1). These eight slots are used by seven applications, one application is executed twice per major cycle in slots C and F. In the characteristics of *Branches*, *Inst fetched*, and *Stores completed*, small spikes can be recognized every millisecond. These spikes stem from a system timer interrupt that is called every millisecond. The interrupt is called continuously even in periods in which it is not clearly visible.

Since the applications are executed independently within their time slots, they do not form a common Fingerprint. Instead, each application has its own Fingerprint and each application/Fingerprint combination needs to be treated separately. This means that, in parallel to the partition switch on the processing core, also the safety-net must change its context to track the next application. As described in Section 6.1.2, the operating system was modified so that the debug interface of the P5020 sends a valid partition id within a Nexus message. The safety-net system changes its internal context each time such a trace message is detected according to the received value which corresponds directly to the newly executed application. This means that the safety-net uses the Fingerprint model that belongs to the currently executed application.

In addition to the pure switching of the model, an application's execution run is not limited to a single slot. This means that an application can be suspended at the end of a slot and resumed at its next slot. The safety-net must take care of this and must continue tracking the progress in the new slot where the execution stopped at the end of the previous slot. As a separate model is created for every partition, it is also possible to independently exchange or modify applications in partitions without having to recreate the model of the other partitions again (incremental development).

In Figure 6.20, the execution characteristic of partition D concatenated over five slots (i.e. major cycles) is shown. It can be recognized that there are performance drops (shown values are the number of completed instructions per time unit) at each partition switch. Since the OS is not configured to provide separate cache areas to the partitions, cache contention between the partitions' time slots occurs that generates these performance drops. It turned out that the Fingerprinting algorithm is robust enough to deal with these short drops without any additional functionality.

The three partitions called *Control*, *Graphics* and *Storage* were used for the evaluation, executed in the slots D, C+F, and H, respectively. 1000 major frame cycles were recorded for the creation of the Fingerprint model in single-core mode. The Fingerprint model is created by first splitting and concatenating individual partitions as described in Section 4.4.4. Both

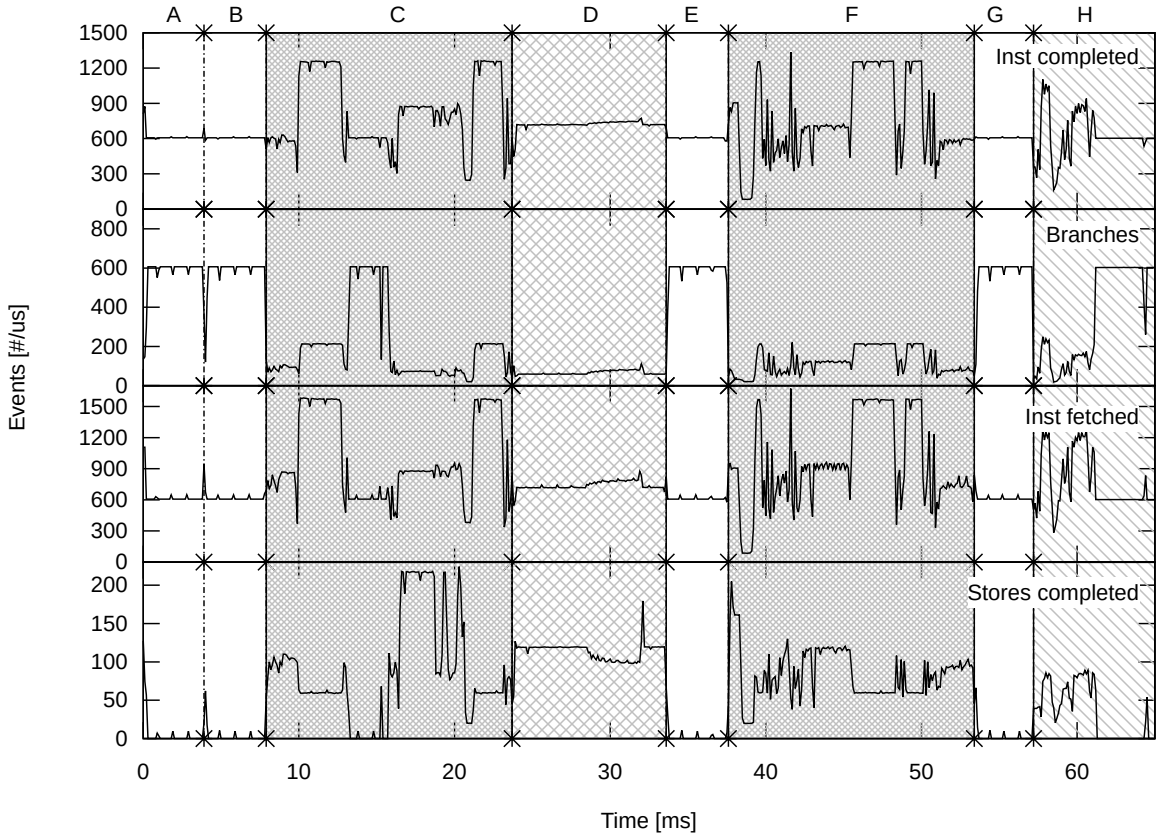


Figure 6.19.: Fingerprint recording of one major cycle of the helicopter IMA application. The partition switches occur at the stars. The slots C and F belong to the same application.

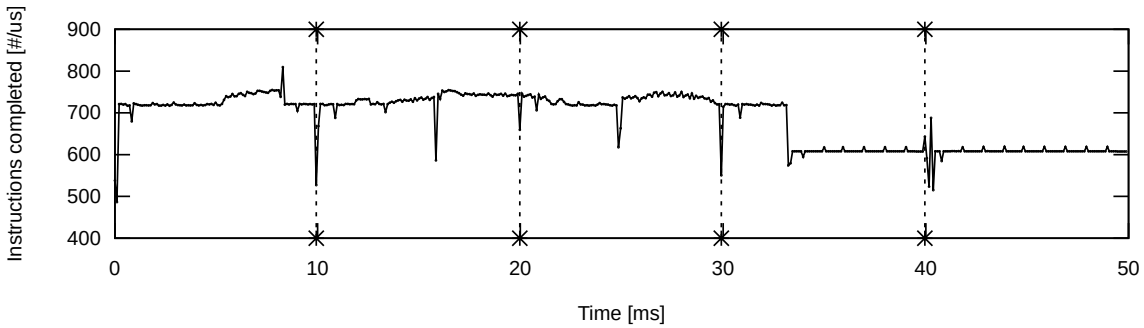
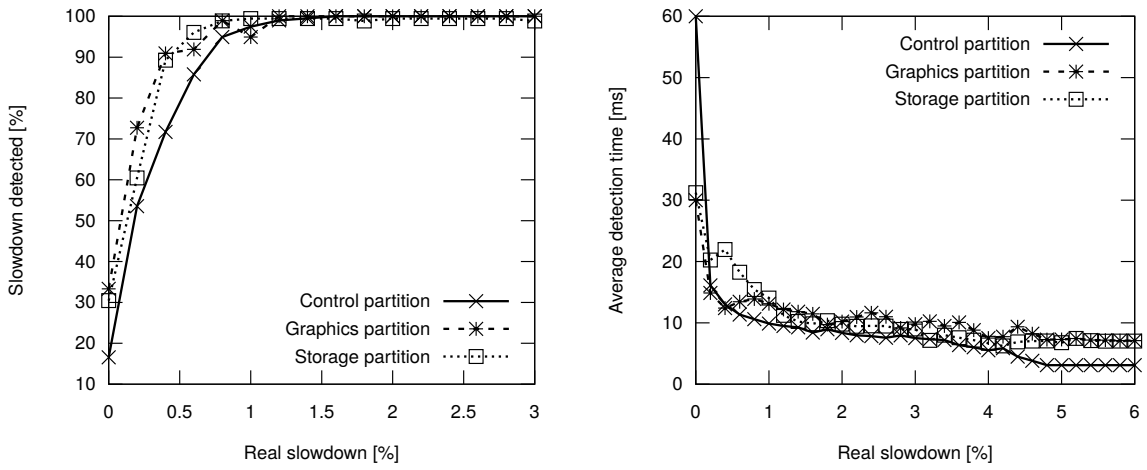


Figure 6.20.: Concatenation of the control partitions extracted from five subsequent major cycles.



(a) Amount of runs in which slowdown is detected for a given simulated slowdown

(b) Average time span until a given slowdown is detected for three selected IMA partitions

Figure 6.21.: Slowdown detection rate and detection time depending on the amount of simulated slowdown.

splitting and concatenation work reliably, and there was no missing ownership trace message observed. After concatenation, the *Control* application has a length of 50 ms (see Figure 6.20), *Graphics* 32 ms and *Storage* 40 ms in every run. Therefore, for example a total amount of 200 different and complete *Control* application periods are used for the model creation. In the second step, the individual application traces are clustered and combined into an individual Fingerprint model for every application as described in Section 5.5.1.

6.5.2. Slowdown Detection

The evaluation is based on a combined real execution and simulation approach. 1000 further major frame cycles were recorded from execution in standalone mode. Interferences between the cores and the resulting slowdown is simulated by shrinking and stretching the recorded curves per partition from 0 % up to 6 %. Shrinking and stretching was done after splitting and concatenation because even if an application is slowed down and executes longer, the partition switch triggered by the operating system is at a defined point in time.

The slowdown ratios that can be reliably detected for the example partitions are shown in Figure 6.21a. For all the partitions, a slowdown above 1.5 % can be detected reliably. However, it can be recognized that the slowdown detection algorithm is behaving slightly differently for different partitions as there are deviations of the curves between 0 % and 1 %. Furthermore, there are false positive detections at a slowdown of 0 %, i.e. a slowdown is detected even though the applications run at full speed.

The average time span until a certain delay is detected is shown in Figure 6.21b. At a slowdown of 0.5 %, the detection algorithm needs an average of about 12 ms to detect the

slowdown for the *control* and *graphics* partition while it needs around 18 ms to detect it for the storage partition. For all the partitions, the optimum is reached at around 4.8 % slowdown. For the *graphics* and *storage* partition, this optimum is at around 7 ms while it is significantly better for the *control* partition (3 ms). There is a deviation because the algorithm responds faster to the characteristics of the *control* partition curves.

This shows that the Fingerprint safety-net approach is suitable for reliably detecting slowdowns of individual partitions in IMA configurations. However, there are false positive slowdown detections even if there is no actual slowdown. This results from the fact that the algorithm extracts data with a certain extraction frequency not synchronized with partition switches. In one execution of a partition, the measurement could be done very close to the partition switch. In this case, all events happening in the partition are counted. In another case, the measurement happens close to 100 μs after the partition switch. In this case, within the first around 100 μs, events are not taken into account and the execution appears to be faster. The algorithm cannot distinguish between real slowdown and this measurement uncertainty and detects it. As the slowdown in such cases is quantified less than 0.1 % and classified as acceptable delay, no countermeasure will be triggered.

The time it takes to detect a slowdown is varying for a given slowdown. For slowdowns from 0 to 3 %, it is not a problem to be detected after a longer period because such small delays should be in the region of the defined acceptable delay. However, a significant delay (greater 3 %) has to be detected early enough before the end of the period of the application. This is the case for all the measured partitions of the system. The *graphics* partition has the smallest period (32 ms) and the worst detection time (7 ms at 5 % slowdown). This detection time is sufficient to perform countermeasures in time. However, if an application has a shorter period, the detection time has to be reduced. This can be done by increasing the sampling frequency but at the cost of higher model sizes and a higher demand for computation in the safety-net processor.

6.6. Discussion of the Results

The results of the evaluation are discussed according to its contribution to the objectives of the safety-net system in Chapter 4.

Objective 1: Keeping the Timing of the Critical Application
The analyses showed that with the virtual timing isolation approach, the deadline of an application can be kept independent of the applications running on the other cores. The complete approach was shown on the example of the TACLeBench suite but the most crucial part, the slowdown detection, was also demonstrated on a real-world helicopter application. It was shown that very small slowdowns (around 1 %) might not be detected. However, this is irrelevant as these small slowdowns are considered in the acceptable delay. An acceptable delay

of 5 % and a deadline placed after 10 % of the single-core WCET is a feasible configuration which leads to a possible utilization of the multicore processor of more than 90 %. For bigger slowdowns (e.g. around 5 %) the deviation of the measured value from the real value was determined to be 2 % in the worst case while being much smaller in the average case (around 1 %). Thus, even in the worst-case uncertainty of the quantification algorithm, the execution is covered by the margin between the acceptable delay and the actual deadline.

It is depended on the application whether the approach is applicable and how the safety-net needs to be implemented. For example, if the application has a very low period ($< 1\,\text{ms}$) the reaction time has to be very high, which leads to a low sample period ($< 32\,\mu\text{s}$). This also implies a need for a faster data link to the processor under observation and a faster safety-net processor compared to an application with $> 10\,\text{ms}$ period where a sample period $> 100\,\mu\text{s}$ is sufficient. The tracking also depends on the performance counter events available on the processor under observation. The evaluation showed that event types that produce continuous event stream with a high number of events per second (e.g. "Instructions completed") lead to the best tracking performance.

The two throttling approaches PWM and frequency scaling can be used to lower the interferences from the low priority cores. However, PWM is more effective and can be used for a more aggressive throttling. Frequency scaling as the only throttling option is not sufficient to safely reduce the interferences.

Objective 2: Efficient Usage of Processing Resources
Three different closed loop controllers were evaluated. All the algorithms satisfied the timing constraints of the critical application. However, the most efficient usage of the other cores was achieved with the progress aware controller. In the evaluation with the *read* benchmark, this controller allowed for a 4 % higher utilization of the other cores compared to the standard proportional controller. For real applications, this advancement is potentially much higher since the progress aware controller only throttles if it is foreseeable that the total acceptable delay will be exceeded. In contrast to that, the other presented controllers also throttle in case the relative slowdown during the program execution is too high even though the absolute acceptable delay can be met.

Objective 3 & 4: Reuse of Existing Single-Core Code and Exchange of Applications without Recertification
The helicopter application was originally developed for a single-core processor and in the evaluation, it was executed on one core of a multicore system. A separate model was created for every partition and the quantification algorithm was executed on that application. It was shown that the approach is applicable for IMA systems and that a real-world avionics application can be observed by the interference quantification algorithm. Since the model was created individually for every partition, it is also possible to exchange independent applications in partitions without having to recreate the model of the other partitions. This is

beneficial for the recertification of an application.

Objective 5: Non-Intrusiveness

The evaluation shows that even though the safety-net approach is not completely non-intrusive, and interference is measurable for very high sample rates, the intrusiveness is very small (<0.2 % for the TACLeBench as well as the *read* benchmark at a sample period of 100 μs) and can be neglected.

7. Conclusion and Future Work

This thesis aims for an efficient usage of multicore processors executing critical real-time applications in avionics while reusing legacy applications without modification. It addresses the problem of WCET overestimation due to the interferences on the shared resources by separately observing the individual applications on the different cores and enforcing the timing of the critical application by an external safety-net processor.

Instead of conducting a WCET analysis taking all the parallel applications into account, a single-core WCET estimate of an application running on one core is taken. An *acceptable delay* is added to account for the interferences on a multicore processor. In order to enforce the timing of the critical application, the progress is measured with Fingerprinting. For that purpose, the performance counters of the critical core are extracted in very small timesteps, which results in a characteristic curve for every execution of the program. The extraction was implemented in this thesis via the high-speed debug interface Aurora and an FPGA external to the observed multicore processor. In standalone mode, without any running applications on the other cores, a model of an application is created from the extracted curves. This is done by recording possible Fingerprints, in the ideal case covering all the program paths. The resulting curves are clustered by a K-Means algorithm and combined to a tree model. During runtime, the extracted performance counter values are compared to the model to determine the progress of the critical application by identifying similarities. Evaluations on a P4080 multicore processor showed that the progress can be measured with a final deviation of less than 1 % for an execution of TACLeBench with running opponent cores.

In case the progress of an application is delayed to an extend that the application cannot finish within the acceptable delay if the interferences continue, the cores creating the interferences are throttled. The identification of the interference creating cores is done similarly to the progress measurement with performance counters of the opponent cores but requires much less extraction bandwidth and computation power on the safety-net device. A controller that takes the progress of an application as well as the time until the final deadline into account was presented. The closer the execution is to the actual deadline, the more aggressive the throttling of the lower critical cores gets. Throttling is either performed by frequency scaling of the interfering cores or by halt and continue with a pulse width modulation scheme. Several experiments were conducted to demonstrate the effectiveness of the different measures. While frequency scaling is less intrusive to the throttled application, it does not guarantee a reduction of interferences. However, it works sufficiently for realistic

applications. Halt and continue can reduce the interferences to zero which was demonstrated on the example of memory intensive opponent threads. The complete control loop was evaluated on a TACLeBench benchmark running on an NXP P4080 multicore processor observed by a Xilinx FPGA implementing a MicroBlaze softcore microcontroller. The timing of the critical application could be kept within the deadline while even for very memory intensive opponent threads, which can be seen as the worst-case applications regarding interferences, the utilization is 64.1 %. For any application that not only executes loads which result in cache misses, the utilization is significantly higher.

7.1. Conclusion

The fundamental approach is to treat the multicore processor as an unsafe device (regarding the timing) that is controlled by a safe external device, the safety-net processor, instead of ensuring intrinsic timing predictability of the multicore. Thus, it does not ensure robust partitioning in the conventional meaning but a virtual timing isolation. The applications on the different cores can influence each other but the external safety-net ensures that the interference on the high critical application is low enough to keep the timing.

The philosophy of ensuring the internal behavior of a multicore processor by an external device complies with the publication on multicore processors of the certification authorities software team (CAST-32A). In the developed design, every critical application is treated individually by relying on individual models recorded in standalone. Thus, the critical as well as the non-critical applications running on the other cores can be exchanged without recreating a Fingerprint model. This eases the porting of legacy applications to the multicore processor and allows the exchange of applications without recertification.

The porting is furthermore eased by the almost non-intrusiveness of the approach. The legacy application does not have to be modified in order to extract the performance counter values. Furthermore, evaluation showed that interference induced by the readout process can be neglected as it is below 0.15 % for access periods >10 μs for a memory intensive benchmark while for a realistic benchmark the measured interference was below 0.06 % for access periods >1 μs.

Partition and task switches can be detected and thus, the safety-net system is applicable to IMA systems. Separate models are created for the different applications running inside an IMA system. The applicability was demonstrated on the example of a real-world helicopter application initially developed as single-core application which consists of eight partitions and was ported to a multicore processor without adding changes to the code for the extraction of performance counters. For all the observed partitions, slowdowns >2 % could be detected in 100 % of the test cases. The approach is not restricted to avionic applications but could also be applied to other high-critical applications, e.g. autonomous driving. However, it can

only be applied to periodic applications as otherwise the creation of a Fingerprint model is not possible.

The approach is designed for the execution of applications of different levels of criticality on the different cores, but observation of multiple critical cores of identical levels of criticality is possible. In case of interferences between several critical cores, throttling is not possible when all cores are of the same level of criticality. In this case, it is possible to switch to a backup system, probably with degraded functionality.

Since the Fingerprinting curves highlight positions in the execution of a program with a high number of accesses to the bus unit interface, it can also be used for the analysis of potential interference positions. Parts of a program with a high amount of memory accesses are identified and applications on other cores can be shifted accordingly during a scheduling process.

7.2. Future Work

The proof of concept of an external virtual timing isolation safety-net is given in this thesis. However, in order to increase practical usage, the approach can be extended and improved as discussed in the following.

In this thesis, the focus of the interferences was on the cores competing for memory bandwidth. However, on-chip master devices like DMA can cause congestion on the memory and interconnect, too. If these devices are used in an operational scenario, these devices have to be throttled similarly to the cores. In order to determine the amount of interferences created by the on-chip devices, the SoC level performance counters might be used similarly to the interference core identification presented in Section 4.6.3. It has to be studied if it is possible to throttle/stop these devices without corrupting data.

The aim of this work was allowing to port legacy applications which were designed for single-core processors to individual cores on a multicore processor. Since currently there are no parallel applications in safety-critical avionics, this topic was not addressed. However, in the future, parallel programs might be developed. It has to be analyzed how a common fingerprint model for parallel threads of the same application can be created or if separate models are a better solution.

In order to allow for a tracking of applications with very low partition sizes ($<1\,\mathrm{ms}$), a faster readout speed is necessary. This can be achieved by a preprocessing of the performance counter values in an FPGA/ASIC, possibly also including a hardware implemented tracking algorithm. The decision for throttling and the throttling itself can be handled by a highly certified microcontroller.

The tracking accuracy can be improved by augmentation of the model with the probability of a slowdown of the respective sample. Since a slowdown can only happen for instruction

sequences that access the interconnect, the bus interface unit accesses performance counter recorded in standalone mode indicates sections which are prone to interferences during runtime. Whenever there is a high amount of bus interface unit accesses, the section can suffer from interferences. For sections with a very low number of cache misses, a slowdown is unlikely. This information can be used to increase the robustness of the tracking.

Depending on the application, the model consumes a high amount of storage in the current implementation. This storage can be reduced by two types of compression: lossless and lossy. A lossless compression could be for example the modeling of loops in the model so that the pattern for one iteration of a loop is stored and reused by the tracking algorithm for all the similar loop runs. Furthermore, a database of similarly shaped parts of the curve can be established and reused in the different paths of the model. Lossy compression methods, e.g. the reduction of bits per sample, could be applied but it has to be evaluated how this affects the accuracy of the tracking. On the other hand, if more complicated compression methods are used, the performance needs of the tracking algorithm are increased.

To demonstrate the practical usage of the approach, it has to be implemented on a highly certifiable microcontroller. The extraction and calculation of the performance counters has to be performed either in the multicore processor or non-intrusively by an FPGA/ASIC acting as the interface between the microcontroller and the multicore processor as explained in Section 4.4.1 because typical microcontrollers are usually not equipped with Aurora or PCIe interfaces needed to extract the performance counter values.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://www.tensorflow.org/

[2] J. Abella, D. Hardy, I. Puaut, E. Quiñones, and F. J. Cazorla, "On the comparison of deterministic and probabilistic WCET estimation techniques," in *2014 26th Euromicro Conference on Real-Time Systems*, July 2014, pp. 266–275.

[3] I. Agirre, J. Abella, M. Azkarate-Askasua, and F. J. Cazorla, "On the Tailoring of CAST-32A Certification Guidance to Real COTS Multicore Architectures," in *12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2017.

[4] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch, "Contention-aware dynamic memory bandwidth isolation with predictability in COTS multicores: An avionics case study," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 2:1–2:22. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7174

[5] Airbus, "Vahana," 2019. [Online]. Available: http://vahana.aero

[6] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo, "Memory-aware scheduling of multicore task sets for real-time systems," in *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2012, pp. 300–309.

[7] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *IEEE Transactions on Computers*, vol. 50, no. 4, pp. 308–321, Apr 2001.

[8] G. Berthon, M. Fumey, X. Jean, H. Misson, L. Mutuel, and D. Regis, "White paper on issues associated with interference applied to multicore processors," 2016.

[9] P. Bieber, F. Boniol, Y. Bouchebaba, J. Brunel, C. Pagetti, O. Poitou, T. Polacsek, L. Santinelli, and N. Sensfelder, "A model-based certification approach for multi/many-core embedded systems," in *9th European Congress Embedded Real Time Software and Systems (ERTS2018)*, 2018.

[10] J. Bin, "Controlling execution time variability using COTS for Safety-critical systems," Theses, Université Paris Sud - Paris XI, Jul. 2014. [Online]. Available: http://tel.archives-ouvertes.fr/tel-01061936

[11] J. Bin, S. Girbal, D. Pérez, A. Grasset, and A. Merigot, "Studying co-running avionic real-time applications on multi-core cots architectures," *Erts2014.Org*, no. August, 2015. [Online]. Available: http://www.erts2014.org/Site/0R4UXE94/Fichier/erts2014_1A2.pdf

[12] Certification Authorities Software Team (CAST), "Position paper cast-32a: Multi-core processors," November 2016. [Online]. Available: http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf

[13] L. Convent, S. Hungerecker, T. Scheffel, M. Schmitz, D. Thoma, and A. Weiss, "Hardware-based runtime verification with embedded tracing units and stream processing," in *Runtime Verification*, C. Colombo and M. Leucker, Eds. Cham: Springer International Publishing, 2018, pp. 43–63.

[14] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Self-tuning schedulers for legacy real-time applications," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 55–68. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755921

[15] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *2012 24th Euromicro Conference on Real-Time Systems*, July 2012, pp. 91–101.

[16] N. Decker, P. Gottschling, C. Hochberger, M. Leucker, T. Scheffel, M. Schmitz, and A. Weiss, "Rapidly adjustable non-intrusive online monitoring for multi-core systems," in *20th Brazilian Symposium on Formal Methods (SBMF)*, vol. LNCS, Springer. Brazil: Springer, 11/2017 2017.

[17] B. Dreyer, C. Hochberger, A. Lange, S. Wegener, and A. Weiss, "Continuous non-intrusive hybrid WCET estimation using waypoint graphs," in *WCET*, 2016.

[18] E. Duesterwald and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *In International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003, pp. 220–231.

[19] U. Durak, J. Becker, S. Hartmann, and N. S. Voros, *Advances in Aeronautical Informatics*. Springer International Publishing, Cham, 2018.

[20] M. Duranton, K. De Bosschere, C. Gamrat, J. Maebe, H. Munk, and O. Zendra, "HiPEAC Vision 2017," 2017.

[21] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *Embedded Real Time Software (ERTS'14)*, TOULOUSE, France, Feb. 2014. [Online]. Available: http://hal.archives-ouvertes.fr/hal-01121700

[22] J. W. Eaton, *GNU Octave*, 2018. [Online]. Available: http://www.gnu.org/software/octave/

[23] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.

[24] Federal Aviation Administration, "Advisory Circular on Society of Automotive Engineers (SAE) Aerospace Recommended Practice (ARP) 4754A," 2011. [Online]. Available: http://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_20-174.pdf

[25] T. Ferrell and U. Ferrell, *Digital Avionics Handbook, Third Edition*, ch. RTCA DO-178B/EUROCAE ED-12B.

[26] S. Fisher, "Certifying applications in a multi-core environment: The world's first multi-core certification to sil 4." SYSGO White Paper, Tech. Rep., 2013.

[27] J. Freitag and S. Uhrig, "Dynamic interference quantification for multicore processors," in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, Sept 2017.

[28] J. Freitag and S. Uhrig, "Closed loop controller for multicore real-time systems," in *Architecture of Computing Systems – ARCS 2018*, M. Berekovic, R. Buchty, H. Hamann, D. Koch, and T. Pionteck, Eds. Cham: Springer International Publishing, 2018, pp. 45–56.

[29] J. Freitag and S. Uhrig, "Quality of service for integrated modular avionics (IMA) on multicore processors using a safety net architecture," in *Proceedings of the 9th European Congress for Embedded Real Time Software and Systems (ERTS), Toulouse*, 2018.

[30] J. Freitag, S. Uhrig, and T. Ungerer, "Virtual Timing Isolation for Mixed-Criticality Systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 13:1–13:23. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/8990

[31] Y. Fu, N. Kottenstette, C. Lu, and X. D. Koutsoukos, "Feedback thermal control of real-time systems on multicore processors," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '12. New York, NY, USA: ACM, 2012, pp. 113–122. [Online]. Available: http://doi.acm.org/10.1145/2380356.2380379

[32] S. Girbal, D. G. Pérez, J. Le Rhun, M. Faugère, C. Pagetti, and G. Durrieu, "A complete toolchain for an interference-free deployment of avionic applications on multi-core systems," in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, Sep. 2015, pp. 7A2–1–7A2–14.

[33] S. Girbal and J. L. Rhun, "BB-RTE: a budget-based runtime engine for mixed and safety critical systems," in *European Congress Embedded Real Time Software and Systems (ERTS2018)*, 2018.

[34] L. Girbinger, "Leveraging on-chip debug units to build dependable computer and monitoring architectures in avionics using modern embedded processors," 2011.

[35] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[36] K. Goossens, M. Koedam, A. Nelson, S. Sinha, S. Goossens, Y. Li, G. Breaban, R. van Kampenhout, R. Tavakoli, J. Valencia, H. A. Balef, B. Akesson, S. Stuijk, M. Geilen, D. Goswami, and M. Nabi, *NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications*. Dordrecht: Springer Netherlands, 2017, pp. 491–530. [Online]. Available: http://doi.org/10.1007/978-94-017-7267-9_17

[37] B. Green, J. Marotta, B. Petre, K. Lillestolen, R. Spencer, N. Gupta, D. O'Leary, J. D. Lee, J. Strasburger, A. Nordsieck, B. Manners, and R. Mahapatra, "Handbook for the selection and evaluation of microprocessors for airborne systems," Federal Aviation Administration - U.S. Department of Transportation, Tech. Rep. DOT/FAA/AR-11/2, 2011. [Online]. Available: http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/AR_11_2.pdf

[38] P. Harrington, *Machine Learning in Action*. Greenwich, CT, USA: Manning Publications Co., 2012.

[39] C. Hernández, J. Abella, F. J. Cazorla, A. Bardizbanyan, J. Andersson, F. Cros, and F. Wartel, "Design and implementation of a time predictable processor: Evaluation with a space case study," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 16:1–16:23. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7173

[40] *The Nexus 5001 Forum - Standard for a Global Embedded Processor Debug Interface*, IEEE Industry Standards and Technology Organization Std. IEEE-ISTO 5001:2012, 2012.

[41] J.-B. Itier, "A380 integrated modular avionics," 2007. [Online]. Available: http://www.artist-embedded.org/docs/Events/2007/IMA/Slides/ARTIST2_IMA_Itier.pdf

[42] M. Jacobs, S. Hahn, and S. Hack, "A framework for the derivation of WCET analyses for multi-core processors," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016, pp. 141–151.

[43] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert, "Ensuring robust partitioning in multicore platforms for IMA systems," in *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, Oct 2012, pp. 7A4–1–7A4–9.

[44] X. Jean, M. Gatti, D. Faura, L. Pautet, and T. Robert, "A software approach for managing shared resources in multicore IMA systems," in *2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)*, Oct 2013, pp. 7D1–1–7D1–15.

[45] T. Kelter, "WCET analysis and optimization for multi-core real-time systems," Ph.D. dissertation, 2015.

[46] T. Kelter and P. Marwedel, "Parallelism analysis: Precise WCET values for complex multi-core systems," in *Formal Techniques for Safety-Critical Systems*, C. Artho and P. C. Ölveczky, Eds. Cham: Springer International Publishing, 2015, pp. 142–158.

[47] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 145–154.

[48] N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, J. H. Anderson, and F. D. Smith, "Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

[49] N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter, "Providing hardware isolation on multicore platforms: What about the operating system?" in *RTAS 2018*, 2018.

[50] R. Kirner and P. Puschner, "Obstacles in worst-case execution time analysis," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, May 2008, pp. 333–339.

[51] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, Canada, April 2019, conference.

[52] B. Koppenhöfer and D. Geiger, "EMC2 use case: Hybrid avionics integrated architecture demonstrator," 2015. [Online]. Available: http://www.artemis-emc2.eu/fileadmin/user_upload/Publications/2015_HiPEAC/EMC2_Bernd_Koppenhoefer_Airbus.pdf

[53] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling, "Multicore in real-time systems – temporal isolation challenges due to shared resources," in *DATE 2013*, 2013.

[54] A. Kritikakou, C. Rochange, M. Faugere, C. Pagetti, M. Roy, S. Girbal, and D. G. Gracia Perez, "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: ACM, 2014, pp. 139:139–139:148. [Online]. Available: http://doi.acm.org/10.1145/2659787.2659799

[55] J. D. Lee, N. Gupta, R. N. Mahapatra, and B. E. Manners, "The FAA handbook on microprocessor selection and evaluation in airborne systems," in *29th Digital Avionics Systems Conference*, Oct 2010, pp. 5.E.4–1–5.E.4–9.

[56] J. Liedtke, H. Haertig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, ser. RTAS '97. Washington, DC,

USA: IEEE Computer Society, 1997, pp. 213–. [Online]. Available: http://dl.acm.org/citation.cfm?id=523983.828369

[57] Lilium , "Lilium Website," 2019. [Online]. Available: http://lilium.com/

[58] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, March 1982.

[59] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. Berkeley, Calif.: University of California Press, 1967, pp. 281–297. [Online]. Available: http://projecteuclid.org/euclid.bsmsp/1200512992

[60] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva, "Power optimization in embedded systems via feedback control of resource allocation," *IEEE Transactions on Control Systems Technology*, vol. 21, no. 1, pp. 239–246, Jan 2013.

[61] R. N. Mahapatra, J. Lee, N. Gupta, and B. Manners, "Microprocessor evaluations for safety-critical, real-time applications: Authority for expenditure no. 43 phase 5 report," no. DOT/FAA/AR-11/5, 2011.

[62] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 45–54.

[63] R. Mancuso, "Next-generation safety-critical systems on multi-core platforms," Ph.D. dissertation, 2017. [Online]. Available: http://hdl.handle.net/2142/97399

[64] G. F. McCormick, *Digital Avionics Handbook, Third Edition*, 2017, ch. Certification of Civil Avionics.

[65] S. Milutinovic, J. Abella, and F. J. Cazorla, "On the assessment of probabilistic WCET estimates reliability for arbitrary programs," *EURASIP Journal on Embedded Systems*, vol. 2017, no. 1, p. 28, Apr 2017. [Online]. Available: http://doi.org/10.1186/s13639-017-0076-8

[66] S. R. M'sirdi, "Modular avionics software integration on multi-core cots : certification-compliant methodology and timing analysis metrics for legacy software reuse in modern aerospace systems," July 2017. [Online]. Available: http://oatao.univ-toulouse.fr/18732/

[67] L. H. Mutuel, X. Jean, and R. Soulat, "Limitations of interference analyses on multicore processors," in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, Sept 2017.

[68] L. H. Mutuel, X. Jean, V. Brindejonc, A. Roger, T. Megel, and E. Alepins, "Assurance of multicore processors in airborne systems," Tech. Rep., 2017.

[69] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *2012 Ninth European Dependable Computing Conference*, May 2012, pp. 132–143.

[70] J. Nowotsch and M. Paulitsch, "Quality of service capabilities for hard real-time applications on multi-core processors," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, ser. RTNS '13. New York, NY, USA: ACM, 2013, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/2516821.2516826

[71] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement." in *ECRTS*. IEEE Computer Society, 2013, pp. 109–118.

[72] NXP Semiconductors, *e500mc Core Reference Manual*, NXP Semiconductors, 2013, rev. 3.

[73] NXP Semiconductors, *e6500 Core Reference Manual*, NXP Semiconductors, 2014, rev. 0. [Online]. Available: http://www.nxp.com/docs/en/reference-manual/E6500RM.pdf

[74] NXP Semiconductors, *P4080 QorIQ Multicore Communication Processor Reference Manual*, rev 2 ed., NXP Semiconductors, 2014.

[75] J. Nyman, "High speed IO using xilinx aurora," Master's thesis, Linköpings universitet, 2013.

[76] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 269–279.

[77] P. J. Prisaznuk, *Digital Avionics Handbook, Third Edition*, 2017, ch. 36, ARINC Specification 653, Avionics Application Software Standard Interface.

[78] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing (Principles, Algorithms, and Applications)*. Prentice-Hall, 2006.

[79] Radio Technical Commission for Aeronautics (RTCA), "DO-254/ED-80, Design Assurance Guidance for Airborne Electronic Hardware," 2000.

[80] Radio Technical Commission for Aeronautics (RTCA), "DO-297/ED-124, Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations," 2005.

[81] Radio Technical Commission for Aeronautics (RTCA), "DO-178C/ED-12C, Software Considerations in Airborne Systems and Equipment Certification," 2011.

[82] Rapita Systems, Ltd., "Multicore Timing Analysis for DO-178C - White Paper," 2019. [Online]. Available: http://www.rapitasystems.com/downloads/multicore-timing-analysis-do-178c

[83] SAE International., "ARP4761 - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment," 1996.

[84] SAE International., "ARP4754A - Guidelines for Development of Civil Aircraft and Systems," 2010.

[85] D. R. Sahoo, S. Swaminathan, R. Al-Omari, M. V. Salapaka, G. Manimaran, and A. K. Somani, "Feedback control for real-time scheduling," in *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, vol. 2, May 2002, pp. 1254–1259 vol.2.

[86] M. Schoeberl, S. Abbaspourseyedi, A. Jordan, E. Kasapaki, W. Puffitsch, J. Sparsø, B. Akesson, N. Audsley, J. Garside, R. Capasso, A. Tocchi, K. Goossens, S. Goossens, Y. Li, S. Hansen, R. Heckmann, S. Hepp, B. Huber, J. Knoop, D. Prokesch, P. Puschner, A. Rocha, and C. Silva, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[87] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn, "Towards a time-predictable dual-issue microprocessor: The patmos approach," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, ser. OASICS 18 Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, Eds., vol. 18. Grenoble, France: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, and Reinhard Wilhelm, Mar. 2011, pp. 11–21. [Online]. Available: http://hal.inria.fr/inria-00585320

[88] L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale, and R. Bradford, "Single core equivalent virtual machines for hard real-time computing on multicore processors," Tech. Rep., 2014.

[89] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatryk, "Fingerprinting: bounding soft-error-detection latency and bandwidth," *IEEE Micro*, vol. 24, no. 6, pp. 22–29, Nov 2004.

[90] C. R. Spitzer, U. Ferrell, and T. Ferrell, *Digital Avionics Handbook, Third Edition*. Taylor and Francis, 2017.

[91] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 62–75.

[92] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937. [Online]. Available: http://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230

[93] S. Uhrig, L. Tadros, and A. Pyka, "MESI-based cache coherence for hard real-time multicore systems," in *Architecture of Computing Systems - ARCS 2015 - 28th International Conference, Porto, Portugal, March 24-27, 2015, Proceedings*, 2015, pp. 212–223. [Online]. Available: http://doi.org/10.1007/978-3-319-16086-3_17

[94] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. G. Zaykov, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quiñones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka, "parMERASA – multi-core execution of parallelised hard real-time applications supporting analysability," in *2013 Euromicro Conference on Digital System Design*, Sept 2013, pp. 363–370.

[95] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Outstanding paper award: Making shared caches more predictable on multicore platforms," in *2013 25th Euromicro Conference on Real-Time Systems*, July 2013, pp. 157–167.

[96] S. Wegener, "Towards Multicore WCET Analysis," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, ser. OpenAccess Series in Informatics (OASIcs), J. Reineke, Ed., vol. 57. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 7:1–7:12. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7311

[97] A. Weiss, "Effiziente externe Beobachtung von CPU-Aktivitäten auf SoCs," Ph.D. dissertation, 2015.

[98] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1347375.1347389

[99] Wind River Systems, Inc., "VxWorks 653 Product Overview," 2018. [Online]. Available: http://www.windriver.com/products/product-overviews/vxworks-653-product-overview-multi-core/

[100] Xilinx, Inc., *7 Series FPGAs Data Sheet: Overview*, DS180 February 27, 2018 ed.

[101] Xilinx, Inc., *Aurora 8B/10B v11.1 IP Product Guide*, PG046 April 4, 2018 ed.

[102] Xilinx, Inc., *AXI 1G/2.5G Ethernet Subsystem v7.1*, PG138 December 5, 2018 ed.

[103] Xilinx, Inc., *MicroBlaze Processor Reference Guide*, UG984 (v2018.2) June 21, 2018 ed.

[104] Xilinx, Inc., *Vivado Design Suite User Guide*, UG973 July 23, 2018 ed.

[105] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 155–166.

[106] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 55–64.

List of Figures

2.1. Evolution of processors in avionics. The past, present and future (from left to right). The gray elements are COTS components while the white hardware components are specifically designed for the individual system. The complexity of the COTS devices is growing from past to future. 5

2.2. Relations between the different guidelines relevant for the certification of systems which include processors and software [84]. 8

2.3. Example of an Integrated Modular Avionics (IMA) system on a single-core processor. Three partitions which include applications of different DALs coexist on one processing core separately from each other. Partitioning is ensured by the ARINC 653 operating system. 9

2.4. Illustration of the partition schedule given in Table 2.2. The major frame is 100 units long. 10

2.5. Example for interference channels on a multicore architecture [68]. In case one core accesses the memory, the requests from the other cores that want to access the memory at the same time are waiting. 12

2.6. Possible versus observed distribution of execution times of an application [10]. The dashed line represents the distribution of the observed times while the solid line shows the distribution of possible execution times. 13

4.1. A comparison of the WCET and the safety-net approach on one period of a high critical application. 28

4.2. Overview block diagram of the safety-net approach which shows the system on chip (SoC) under observation by the safety-net processor. The different building blocks are explained in the corresponding sections as denoted in brackets. 29

4.3. Fingerprint curve with the event counter *Instructions Completed* of two executions of the TACLeBench benchmark suite. The solid line shows a run without co-running applications while the dashed line shows a 4 % delayed execution that suffers from interferences created by applications on the other cores. The arrows indicate the corresponding delayed execution. 31

4.4. Measured fingerprint curves of the four event counters with a sample period of 100 μs: *Instructions completed* on the top curve, *Instructions fetched* second, *Branches completed* third, and *Stores completed* in lowest curve when executing the sequential benchmarks of the TACLeBench [23] benchmark suite (Table A.1). 32

4.5. Different possible safety-net architectures. 35

4.6. Factors depending on the sample rate. The solid line represents a low sample rate while the dashed line shows a high sample rate. The further the line is at the edge, the more beneficial for the corresponding property. For example, a low sample rate is beneficial for bandwidth requirements, processing demand and the model size, but it leads to a less precise model and a longer reaction time. 36
4.7. Partition or task switches and the respective ownership trace messages (OTM) are not aligned with the measurement periods. Measurement 2, which is directly following the task/partition switch has to be ignored because it is not certain which portion of the measurement has to be assigned to task 1, task 2 and the scheduler. 38
4.8. Splitting and concatenation of fingerprints resulting from IMA partitions. . . 39
4.9. Fingerprint recording for ten different runs with the same input parameters of a real avionics application. 41
4.10. Generation process of the Fingerprint model. The raw data (left) is clustered by a bisecting k-means algorithm and is reduced to the median curve to build up the Fingerprint model. 41
4.11. Example of a *Fingerprint* model implemented as a tree model during model creation and tracking. 45
4.12. Curve tracking for a different number of simultaneously compared samples (δ). 50
4.13. Different throttling techniques applicable to modern multicore processors. . . 53
4.14. Examples for different controller algorithms. 55
4.15. Block diagram of the closed loop. The senor elements are the Fingerprinting for the critical application and the interference detection for the other cores in the feedback loop. 56
4.16. The throttling u of a low priority core for a varying interference core detection i for different values of total throttling u_{total} according to Equation 4.12. . . . 58

5.1. Interfaces of the hardware setup used for implementation. The NXP P4080 multicore processor is connected to the timing isolation system implemented in a Xilinx Virtex 7 FPGA via a standardized Aurora trace port. 60
5.2. Block diagram of the NXP P4080 [74]. The main sources of interference are the central interconnect and the memory controllers. 61
5.3. Block diagram of the logic implemented in the FPGA. 63
5.4. Extraction process of the performance counters in the cores and possible interference channels (dashed line) on the example of one core of the NXP P4080. 66
5.5. Model creation process including the used tools. After the trace is split into different streams for the different threads (Split Traces at OTM) the following steps are individually performed for every thread. 68
5.6. Example of a Fingerprint model encoded as adjacency matrix with edge probabilities. 70
5.7. Flowchart of the software executed on safety-net processor. The loop after the initialization is running in a 100 μs period. 71
5.8. Clock selection infrastructure of the NXP P4080 [74]. The selection of the clocks can be done by the RCPM during runtime, individually per core. . . . 73

6.1. Pilot interface of a helicopter application. Terrain that is higher than the current altitude is highlighted. The current position of the helicopter is displayed in the center of the picture. [52] 76
6.2. Hybrid environment with the data generation on the P4080, extracted by the Virtex-7 and forwarded to a workstation. The data processing is done offline on the workstation. 79
6.3. Block diagram of the safety-net hardware setup used for the P4080 development system in the full system integration environment. 80
6.4. Block diagram of the safety-net hardware setup used for the helicopter application IMA system [52]. 81
6.5. Reaction time and model size of the interference quantification algorithm depending on the sample period. The experiment is based on the *dynamic TACLeBench* execution with the *realistic* cache configuration and four *read* opponent threads on the other cores. The tracked performance counter event is *Instructions completed*. 82
6.6. Interference quantification accuracy depending on the performance counter events used for the tracking. The experiment is based on the *static TACLeBench* execution with the *realistic* cache configuration, four *read* opponent threads on the other cores, and a sample period of 100 μs. 83
6.7. Comparison of the *Instructions completed* and *FPU finish* performance counter events of one run of the *static TACLeBench* at a sample period of 100 μs. . . 84
6.8. Quality of the quantification over the runtime of the static TACLeBench suite in standalone and with seven opponent cores (*Slowdown*) using the *Realistic* cache configuration (L1 cache is enabled). The plotted dots represent the mean values while the bars reflect the minimum and maximum value measured. The standalone executions end at 48.5 ms while the slowed down executions end at 52.4 ms, which is a total slowdown of around 8 %. 86
6.9. Frequency Scaling and PWM efficiency with the read algorithm (see Chapter 6.1.3) on core 0 as well as read opponents on the other seven cores. These measurements were conducted with the *Maximum interference* cache configuration (all caches disabled). A separate analysis (not shown here) indicated that the curves are very similar for enabled local caches as the read algorithm is designed to cause maximum stress on the interconnect and does not take advantage of caches. 88
6.10. Frequency scaling and PWM efficiency with TACLeBench on core 0 and write opponents on the other cores. For Figure **a** and **b** the *Maximum interference* cache configuration applies while for Figure **c** and **d** the *Realistic* cache configuration (L1 cache enabled) was used. 89
6.11. Frequency scaling and PWM efficiency with TACLe on core 0 and TACLe opponents on the other cores. The measurements were taken with the *Maximum interference* cache configuration. 90
6.12. Threshold-based controller with halt and continue. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores. 92
6.13. Proportional controller with PWM. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores. . . 93

6.14. Proportional controller with frequency scaling. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores. 94
6.15. Progress aware controller with PWM. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores. 95
6.16. Progress aware controller with frequency scaling. TACLeBench performance over time with and without throttling. *Write* opponents are executed on the other cores. 96
6.17. Slowdown of the TACLeBench execution on one core depending on the access period of the performance counter read-out process. 98
6.18. Slowdown of the read benchmark to the on-chip SRAM (L3 cache configured as SRAM) while the L1 and L2 caches are disabled. 98
6.19. Fingerprint recording of one major cycle of the helicopter IMA application. The partition switches occur at the stars. The slots C and F belong to the same application. 101
6.20. Concatenation of the control partitions extracted from five subsequent major cycles. 101
6.21. Slowdown detection rate and detection time depending on the amount of simulated slowdown. 102

List of Tables

2.1. Design Assurance Levels (DAL) according to [90]. The probability defines the chance of failure per flight hour. 8
2.2. Example of an IMA partition schedule. 10
2.3. Summary of CAST-32A objectives [3, 12]. 15

6.1. List of partitions scheduled on the helicopter application including duration and maximum number of observed memory accesses as published by Agrawal et al. [4]. 77
6.2. Different cache configurations used in the evaluations. 78

A.1. Selection of 19 of the 23 TACLeBench sequential benchmarks selected to be used in this thesis. (published in [23]) 131

Acronyms

AES	Advanced Encryption Standard.
AFDX	Avionics Full-Duplex Switched Ethernet.
AMP	Asymmetric Multiprocessing.
APEX	APplication EXecutive.
ARINC	Aeronautical Radio, Incorporated.
ASIC	Application-Specific Integrated Circuit.
AXI	Advanced Extensible Interface.
BIU	Bus Interface Unit.
CAST	Certification Authorities Software Team.
COTS	Commercial Off-The-Shelf.
CPU	Central Processing Unit.
DAL	Design Assurance Level.
DMA	Direct Memory Access.
EASA	European Aviation Safety Agency.
EUROCAE	European Organization for Civil Aviation Equipment.
FAA	Federal Aviation Administration.
FIFO	First In First Out.
FMC	FPGA Mezzanine Card.
FPGA	Field Programmable Gate Array.
FPU	Floating Point Unit.
GPIO	General-Purpose Input/Output.
GPU	Graphics Processing Unit.
HSSTP	High Speed Serial Trace Probe.
HSTP	High-Speed Transport Protocol.
I2C	Inter Integrated Circuit.
IMA	Integrated Modular Avionics.

IP	Intellectual Property.
JTAG	Joint Test Action Group.
L1	Level-1 Cache.
L2	Level-2 Cache.
L3	Level-3 Cache.
MAC	Media Access Control.
MCDC	Modified Condition/Decision Coverage.
MCP	Multicore Processor.
MDO	Message Data Output.
MMU	Memory Management Unit.
MPSoC	Multi-Processor System-on-Chip.
MPU	Memory Protection Unit.
MSE	Message Start/End.
NoC	Network-on-Chip.
NPIDR	Nexus Process Id Register.
OS	Operating System.
OTM	Ownership Trace Message.
PAMU	Peripheral Access Management Unit.
PCIe	Peripheral Component Interconnect Express.
PLL	Phase-Locked Loop.
PMC	Performance Monitor Counter.
PWM	Pulse Width Modulation.
RCPM	Run Control and Power Management.
RISC	Reduced Instruction Set Computer.
RTCA	Radio Technical Commission for Aeronautics.
SCE	Single Core Equivalence.
SDRAM	Synchronous Dynamic Random Access Memory.
SerDes	Serializer/Deserializer.
SEU	Single Event Upset.
SMP	Symmetric Multiprocessing.
SoC	System-on-Chip.
SRAM	Static Random Access Memory.
SWaP	Space, Weight and Power.

TACLe	TACLe Benchmarks.
TACLeBench	TACLe Benchmarks.
TDMA	Time-Division Multiple Access.
TFTP	Trivial File Transfer Protocol.
UART	Universal Asynchronous Receiver/Transmitter.
VHDL	VHSIC Hardware Description Language.
WCET	Worst-Case Execution Time.

A. TACLeBench

Name	Description	Code Size (SLOC)	Origin
adpcm_dec	ADPCM decoder	293	SNU-RT
adpcm_enc	ADPCM encoder	316	SNU-RT
anagram	Word anagram computation	2710	Raymond Chen
audiobeam	Audio beam former	833	StreamIt
cjpeg_transupp	JPEG image transcoding routines	608	MediaBench
cjpeg_wrbmp	JPEG image bitmap writing code	892	Thomas G. Lane
epic	Efficient pyramid image coder	451	MediaBench
fmref	Software FM radio with equalizer	680	StreamIt
g723_enc	CCITT G.723 encoder	480	SUN Microsystems
gsm_dec	GSM provisional standard decoder	543	MediaBench
gsm_enc	GSM provisional standard encoder	1491	MediaBench
h264_dec	H.264 block decoding functions	460	MediaBench
huff_dec	Huffman decoding with a file source to decompress	183	David Bourgin
huff_enc	Huffman encoding with a file source to compress	325	David Bourgin
ndes	Complex embedded code	260	MRTC
petrinet	Petri net simulation	500	Friedhelm Stappert
rijndael_dec	Rijndael AES decryption	820	MiBench
rijndael_enc	Rijndael AES encryption	734	MiBench
statemate	Statechart simulation of a car window lift control	1038	Friedhelm Stappert

Table A.1.: Selection of 19 of the 23 TACLeBench sequential benchmarks selected to be used in this thesis. (published in [23])